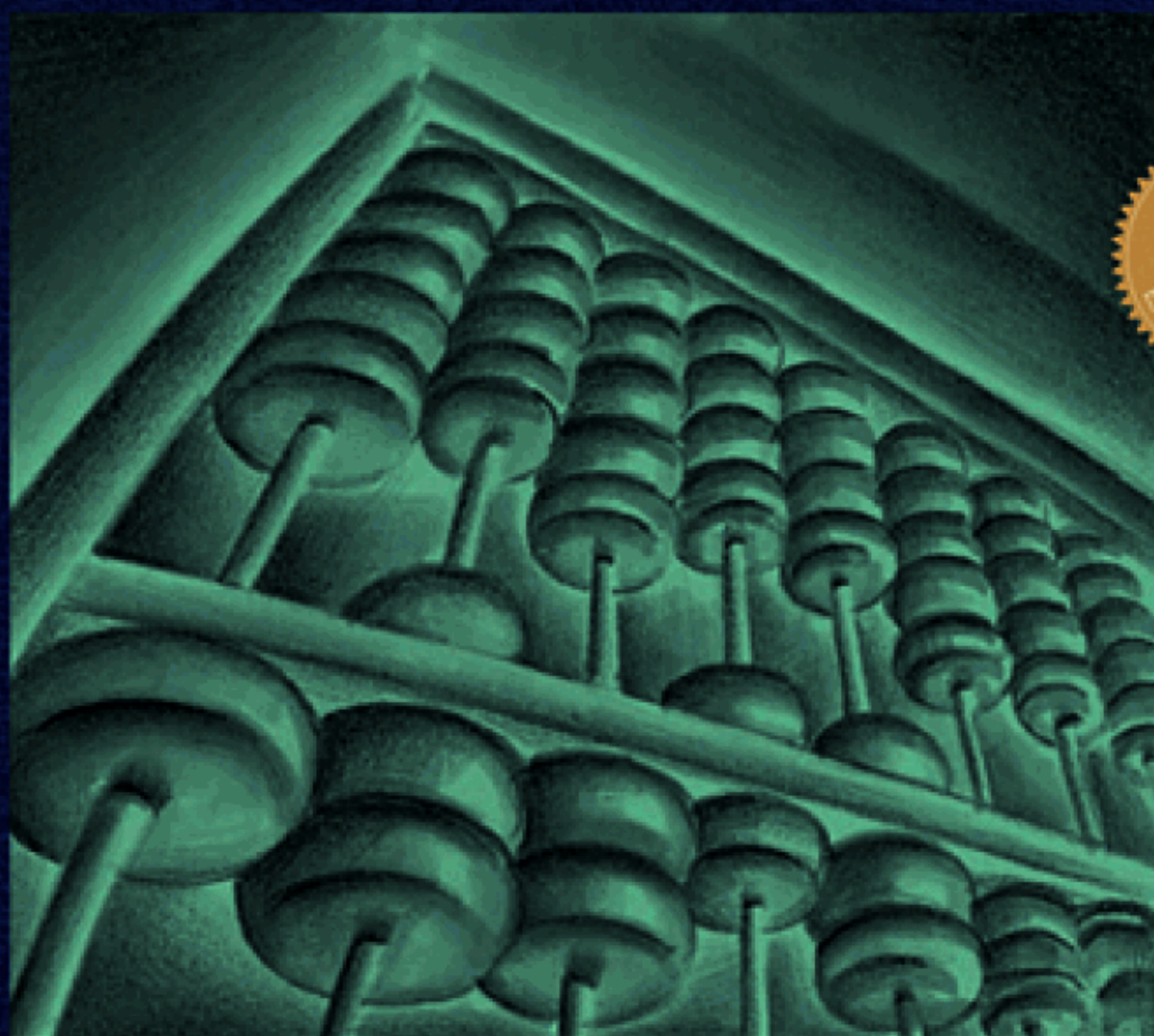


COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE



THIRD
3
EDITION

DAVID A. PATTERSON
JOHN L. HENNESSY



T H I R D E D I T I O N

Computer Organization Design

T H E H A R D W A R E / S O F T W A R E I N T E R F A C E

ACKNOWLEDGEMENTS

Figures 1.9, 1.15 Courtesy of Intel.

Figure 1.11 Courtesy of Storage Technology Corp.

Figures 1.7.1, 1.7.2, 6.13.2 Courtesy of the Charles Babbage Institute, University of Minnesota Libraries, Minneapolis.

Figures 1.7.3, 6.13.1, 6.13.3, 7.9.3, 8.11.2 Courtesy of IBM.

Figure 1.7.4 Courtesy of Cray Inc.

Figure 1.7.5 Courtesy of Apple Computer, Inc.

Figure 1.7.6 Courtesy of the Computer History Museum.

Figure 7.33 Courtesy of AMD.

Figures 7.9.1, 7.9.2 Courtesy of Museum of Science, Boston.

Figure 7.9.4 Courtesy of MIPS Technologies, Inc.

Figure 8.3 ©Peg Skorpinski.

Figure 8.11.1 Courtesy of the Computer Museum of America.

Figure 8.11.3 Courtesy of the Commercial Computing Museum.

Figures 9.11.2, 9.11.3 Courtesy of NASA Ames Research Center.

Figure 9.11.4 Courtesy of Lawrence Livermore National Laboratory.

Computers in the Real World:

Photo of “A Laotian villager,” courtesy of David Sanger.

Photo of an “Indian villager,” property of Encore Software, Ltd., India.

Photos of “Block and students” and “a pop-up archival satellite tag,” courtesy of Professor Barbara Block. Photos by Scott Taylor.

Photos of “Professor Dawson and student” and “the Mica micromote,” courtesy of AP/World Wide Photos.

Photos of “images of pottery fragments” and “a computer reconstruction,” courtesy of Andrew Willis and David B. Cooper, Brown University, Division of Engineering.

Photo of “the Eurostar TGV train,” by Jos van der Kolk.

Photo of “the interior of a Eurostar TGV cab,” by Andy Veitch.

Photo of “firefighter Ken Whitten,” courtesy of World Economic Forum.

Graphic of an “artificial retina,” © The San Francisco Chronicle. Reprinted by permission.

Image of “A laser scan of Michelangelo’s statue of David,” courtesy of Marc Levoy and Dr. Franca Falletti, director of the Galleria dell’Accademia, Italy.

“An image from the Sistine Chapel,” courtesy of Luca Pezzati. IR image recorded using the scanner for IR reflectography of the INOA (National Institute for Applied Optics, <http://arte.ino.it>) at the Opificio delle Pietre Dure in Florence.

T H I R D E D I T I O N

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

David A. Patterson

University of California, Berkeley

John L. Hennessy

Stanford University

With a contribution by

Peter J. Ashenden

Ashenden Designs Pty Ltd

James R. Larus

Microsoft Research

Daniel J. Sorin

Duke University



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier



MORGAN KAUFMANN PUBLISHERS

Senior Editor	Denise E. M. Penrose
Publishing Services Manager	Simon Crump
Editorial Assistant	Summer Block
Cover Design	Ross Caron Design
Cover and Chapter Illustration	Chris Asimoudis
Text Design	GGs Book Services
Composition	Nancy Logan and Dartmouth Publishing, Inc.
Technical Illustration	Dartmouth Publishing, Inc.
Copyeditor	Ken DellaPenta
Proofreader	Jacqui Brownstein
Indexer	Linda Buskus
Interior printer	Courier
Cover printer	Courier

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

© 2005 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>) by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Application submitted

ISBN: 1-55860-604-1

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com.

Printed in the United States of America
04 05 06 07 08 5 4 3 2 1


Contents

Preface ix

CHAPTERS

1

Computer Abstractions and Technology 2


- 1.1 Introduction 3
- 1.2 Below Your Program 11
- 1.3 Under the Covers 15
- 1.4 Real Stuff: Manufacturing Pentium 4 Chips 28
- 1.5 Fallacies and Pitfalls 33
- 1.6 Concluding Remarks 35
-  1.7 Historical Perspective and Further Reading 36
- 1.8 Exercises 36



COMPUTERS IN THE REAL WORLD

Information Technology for the 4 Billion without IT 44

2


Instructions: Language of the Computer 46

- 2.1 Introduction 48
- 2.2 Operations of the Computer Hardware 49
- 2.3 Operands of the Computer Hardware 52
- 2.4 Representing Instructions in the Computer 60
- 2.5 Logical Operations 68
- 2.6 Instructions for Making Decisions 72
- 2.7 Supporting Procedures in Computer Hardware 79
- 2.8 Communicating with People 90
- 2.9 MIPS Addressing for 32-Bit Immediates and Addresses 95
- 2.10 Translating and Starting a Program 106
- 2.11 How Compilers Optimize 116
-  2.12 How Compilers Work: An Introduction 121

	2.13 A C Sort Example to Put It All Together	121
	2.14 Implementing an Object-Oriented Language	130
	2.15 Arrays versus Pointers	130
	2.16 Real Stuff: IA-32 Instructions	134
	2.17 Fallacies and Pitfalls	143
	2.18 Concluding Remarks	145
	2.19 Historical Perspective and Further Reading	147
	2.20 Exercises	147


COMPUTERS IN THE REAL WORLD
Helping Save Our Environment with Data 156

3 Arithmetic for Computers 158

	3.1 Introduction	160
	3.2 Signed and Unsigned Numbers	160
	3.3 Addition and Subtraction	170
	3.4 Multiplication	176
	3.5 Division	183
	3.6 Floating Point	189
	3.7 Real Stuff: Floating Point in the IA-32	217
	3.8 Fallacies and Pitfalls	220
	3.9 Concluding Remarks	225
	3.10 Historical Perspective and Further Reading	229
	3.11 Exercises	229




COMPUTERS IN THE REAL WORLD
Reconstructing the Ancient World 236

4 Assessing and Understanding Performance 238



	4.1 Introduction	240
	4.2 CPU Performance and Its Factors	246
	4.3 Evaluating Performance	254
	4.4 Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors	259
	4.5 Fallacies and Pitfalls	266
	4.6 Concluding Remarks	270
	4.7 Historical Perspective and Further Reading	272
	4.8 Exercises	272

COMPUTERS IN THE REAL WORLD
Moving People Faster and More Safely 280

5**The Processor: Datapath and Control 282**


- 5.1 Introduction 284
- 5.2 Logic Design Conventions 289
- 5.3 Building a Datapath 292
- 5.4 A Simple Implementation Scheme 300
- 5.5 A Multicycle Implementation 318
- 5.6 Exceptions 340
-  5.7 Microprogramming: Simplifying Control Design 346
-  5.8 An Introduction to Digital Design Using a Hardware Design Language 346
- 5.9 Real Stuff: The Organization of Recent Pentium Implementations 347
- 5.10 Fallacies and Pitfalls 350
- 5.11 Concluding Remarks 352
-  5.12 Historical Perspective and Further Reading 353
- 5.13 Exercises 354

COMPUTERS IN THE REAL WORLD**Empowering the Disabled 366****6****Enhancing Performance with Pipelining 368**



- 6.1 An Overview of Pipelining 370
- 6.2 A Pipelined Datapath 384
- 6.3 Pipelined Control 399
- 6.4 Data Hazards and Forwarding 402
- 6.5 Data Hazards and Stalls 413
- 6.6 Branch Hazards 416
-  6.7 Using a Hardware Description Language to Describe and Model a Pipeline 426
- 6.8 Exceptions 427
- 6.9 Advanced Pipelining: Extracting More Performance 432
- 6.10 Real Stuff: The Pentium 4 Pipeline 448
- 6.11 Fallacies and Pitfalls 451
- 6.12 Concluding Remarks 452
-  6.13 Historical Perspective and Further Reading 454
- 6.14 Exercises 454

COMPUTERS IN THE REAL WORLD**Mass Communication without Gatekeepers 464**

7**Large and Fast: Exploiting Memory Hierarchy 466**

- 7.1 Introduction 468
- 7.2 The Basics of Caches 473
- 7.3 Measuring and Improving Cache Performance 492
- 7.4 Virtual Memory 511
- 7.5 A Common Framework for Memory Hierarchies 538
- 7.6 Real Stuff: The Pentium P4 and the AMD Opteron Memory Hierarchies 546
- 7.7 Fallacies and Pitfalls 550
- 7.8 Concluding Remarks 552
-  7.9 Historical Perspective and Further Reading 555
- 7.10 Exercises 555

COMPUTERS IN THE REAL WORLD**Saving the World's Art Treasures 562****8****Storage, Networks, and Other Peripherals 564**

- 8.1 Introduction 566
- 8.2 Disk Storage and Dependability 569
-  8.3 Networks 580
- 8.4 Buses and Other Connections between Processors, Memory, and I/O Devices 581
- 8.5 Interfacing I/O Devices to the Processor, Memory, and Operating System 588
- 8.6 I/O Performance Measures: Examples from Disk and File Systems 597
- 8.7 Designing an I/O System 600
- 8.8 Real Stuff: A Digital Camera 603
- 8.9 Fallacies and Pitfalls 606
- 8.10 Concluding Remarks 609
-  8.11 Historical Perspective and Further Reading 611
- 8.12 Exercises 611

COMPUTERS IN THE REAL WORLD**Saving Lives through Better Diagnosis 622****9****Multiprocessors and Clusters 9-2**

- 9.1 Introduction 9-4
- 9.2 Programming Multiprocessors 9-8
- 9.3 Multiprocessors Connected by a Single Bus 9-11

- 9.4 Multiprocessors Connected by a Network 9-20
- 9.5 Clusters 9-25
- 9.6 Network Topologies 9-27
- 9.7 Multiprocessors Inside a Chip and Multithreading 9-30
- 9.8 Real Stuff: The Google Cluster of PCs 9-34
- 9.9 Fallacies and Pitfalls 9-39
- 9.10 Concluding Remarks 9-42
- 9.11 Historical Perspective and Further Reading 9-47
- 9.12 Exercises 9-55

A P P E N D I C E S



Assemblers, Linkers, and the SPIM Simulator A-2

- A.1 Introduction A-3
- A.2 Assemblers A-10
- A.3 Linkers A-18
- A.4 Loading A-19
- A.5 Memory Usage A-20
- A.6 Procedure Call Convention A-22
- A.7 Exceptions and Interrupts A-33
- A.8 Input and Output A-38
- A.9 SPIM A-40
- A.10 MIPS R2000 Assembly Language A-45
- A.11 Concluding Remarks A-81
- A.12 Exercises A-82



The Basics of Logic Design B-2

- B.1 Introduction B-3
- B.2 Gates, Truth Tables, and Logic Equations B-4
- B.3 Combinational Logic B-8
- B.4 Using a Hardware Description Language B-20
- B.5 Constructing a Basic Arithmetic Logic Unit B-26
- B.6 Faster Addition: Carry Lookahead B-38
- B.7 Clocks B-47
- B.8 Memory Elements: Flip-flops, Latches, and Registers B-49
- B.9 Memory Elements: SRAMs and DRAMs B-57
- B.10 Finite State Machines B-67
- B.11 Timing Methodologies B-72

B.12 Field Programmable Devices B-77

B.13 Concluding Remarks B-78

B.14 Exercises B-79



Mapping Control to Hardware C-2

C.1 Introduction C-3

C.2 Implementing Combinational Control Units C-4

C.3 Implementing Finite State Machine Control C-8

C.4 Implementing the Next-State Function with a Sequencer C-21

C.5 Translating a Microprogram to Hardware C-27

C.6 Concluding Remarks C-31

C.7 Exercises C-32



A Survey of RISC Architectures for Desktop, Server, and Embedded Computers D-2

D.1 Introduction D-3

D.2 Addressing Modes and Instruction Formats D-5

D.3 Instructions: The MIPS Core Subset D-9

D.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs D-16

D.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs D-19

D.6 Instructions: Common Extensions to MIPS Core D-20

D.7 Instructions Unique to MIPS64 D-25

D.8 Instructions Unique to Alpha D-27

D.9 Instructions Unique to SPARC v.9 D-29

D.10 Instructions Unique to PowerPC D-32

D.11 Instructions Unique to PA-RISC 2.0 D-34

D.12 Instructions Unique to ARM D-36

D.13 Instructions Unique to Thumb D-38

D.14 Instructions Unique to SuperH D-39

D.15 Instructions Unique to M32R D-40

D.16 Instructions Unique to MIPS16 D-41

D.17 Concluding Remarks D-43

D.18 Acknowledgments D-46

D.19 References D-47

Index I-1

 Glossary G-1

 Further Reading FR-1

Preface

*The most beautiful thing we can experience is the mysterious.
It is the source of all true art and science.*

Albert Einstein, *What I Believe*, 1930

About This Book

We believe that learning in computer science and engineering should reflect the current state of the field, as well as introduce the principles that are shaping computing. We also feel that readers in every specialty of computing need to appreciate the organizational paradigms that determine the capabilities, performance, and, ultimately, the success of computer systems.

Modern computer technology requires professionals of every computing specialty to understand both hardware and software. The interaction between hardware and software at a variety of levels also offers a framework for understanding the fundamentals of computing. Whether your primary interest is hardware or software, computer science or electrical engineering, the central ideas in computer organization and design are the same. Thus, our emphasis in this book is to show the relationship between hardware and software and to focus on the concepts that are the basis for current computers.

The audience for this book includes those with little experience in assembly language or logic design who need to understand basic computer organization as well as readers with backgrounds in assembly language and/or logic design who want to learn how to design a computer or understand how a system works and why it performs as it does.

About the Other Book

Some readers may be familiar with *Computer Architecture: A Quantitative Approach*, popularly known as Hennessy and Patterson. (This book in turn is called Patterson and Hennessy.) Our motivation in writing that book was to describe the principles of computer architecture using solid engineering funda-

mentals and quantitative cost/performance trade-offs. We used an approach that combined examples and measurements, based on commercial systems, to create realistic design experiences. Our goal was to demonstrate that computer architecture could be learned using quantitative methodologies instead of a descriptive approach. It is intended for the serious computing professional who wants a detailed understanding of computers.











































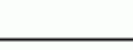






































A majority of the readers for this book do not plan to become computer architects. The performance of future software systems will be dramatically affected, however, by how well software designers understand the basic hardware techniques at work in a system. Thus, compiler writers, operating system designers, database programmers, and most other software engineers need a firm grounding in the principles presented in this book. Similarly, hardware designers must understand clearly the effects of their work on software applications.


Thus, we knew that this book had to be much more than a subset of the material in *Computer Architecture*, and the material was extensively revised to match the different audience. We were so happy with the result that the subsequent editions of *Computer Architecture* were revised to remove most of the introductory material; hence, there is much less overlap today than with the first editions of both books.


Changes for the Third Edition


We had six major goals for the third edition of *Computer Organization and Design*: make the book work equally well for readers with a software focus or with a hardware focus; improve pedagogy in general; enhance understanding of program performance; update the technical content to reflect changes in the industry since the publication of the second edition in 1998; tie the ideas from the book more closely to the real world *outside* the computing industry; and reduce the size of this book.


First, the table on the next page shows the hardware and software paths through the material. Chapters 1, 4, and 7 are found on both paths, no matter what the experience or the focus. Chapters 2 and 3 are likely to be review material for the hardware-oriented, but are essential reading for the software-oriented, especially for those readers interested in learning more about compilers and object-oriented programming languages. The first sections of Chapters 5 and 6 give overviews for those with a software focus. Those with a hardware focus, however, will find that these chapters present core material; they may also, depending on background, want to read Appendix B on logic design first and the sections on microprogramming and how to use hardware description languages to specify control. Chapter 8 on input/output is key to readers with a software focus and should be read if time permits by others. The last chapter on multiprocessors and clusters is again a question of time for the reader. Even the history sections show this balanced focus; they include short histories of programming languages, compilers, numerical software, operating systems, networking protocols, and databases.


Chapter or Appendix	Sections	Software Focus	Hardware Focus
1. Computer Abstractions and Technology	1.1 to 1.6		
	 1.7 (History)		
2. Instructions: Language of the Computer	2.1 to 2.11		
	 2.12 (Compilers)		
	2.13 (C sort)		
	 2.14 (Java)		
	2.15 to 2.18		
	 2.19 (History)		
3. Arithmetic for Computers	3.1 to 3.11		
	 3.12 (History)		
D. RISC instruction set architectures	 D.1 to D.19		
4. Assessing and Understanding Performance	4.1 to 4.6		
	 4.7 (History)		
B. The Basics of Logic Design	 B.1 to B.13		
5. The Processor: Datapath and Control	5.1 (Overview)		
	5.2 to 5.7		
	 5.8 (Microcode)		
	 5.9 (Verilog)		
	5.10 to 5.12		
	 5.13 (History)		
C. Mapping Control to Hardware	 C.1 to C.6		
6. Enhancing Performance with Pipelining	6.1 (Overview)		
	6.2 to 6.6		
	 6.7 (verilog)		
	6.8 to 6.9		
	6.10 to 6.12		
	 6.13 (History)		
7. Large and Fast: Exploiting Memory Hierarchy	7.1 to 7.8		
	 7.9 (History)		
8. Storage, Networks, and Other Peripherals	8.1 to 8.2		
	 8.3 (Networks)		
	8.4 to 8.10		
	 8.13 (History)		
9. Multiprocessors and Clusters	 9.1 to 9.10		
	 9.11 (History)		
A. Assemblers, Linkers, and the SPIM Simulator	 A.1 to A.12		
Computers in the Real World	Between Chapters		

Read carefully 

Read if have time 

Reference 

Review or read 

Read for culture 

The next goal was to improve the exposition of the ideas in the book, based on difficulties mentioned by readers of the second edition. We added five new book elements to help. To make the book work better as a reference, we placed definitions of new terms in the margins at their first occurrence. We hope this will help readers find the sections when they want to refer back to material they have already read. Another change was the insertion of the “Check Yourself” sections, which we added to help readers to check their comprehension of the material on the first time through it. A third change is that added extra exercises in the “For More Practice” section. Fourth, we added the answers to the “Check Yourself” sections and to the For More Practice exercises to help readers see for themselves if they understand the material by comparing their answers to the book. The final new book element was inspired by the “Green Card” of the IBM System/360. We believe that you will find that the MIPS Reference Data Card will be a handy reference when writing MIPS assembly language programs. Our idea is that you will remove the card from the front of the book, fold it in half, and keep it in your pocket, just as IBM S/360 programmers did in the 1960s.

Third, computers are so complex today that understanding the performance of a program involves understanding a good deal about the underlying principles and the organization of a given computer. Our goal is that readers of this book should be able to understand the performance of their programs and how to improve it. To aid in that goal, we added a new book element called “Understanding Program Performance” in several chapters. These sections often give concrete examples of how ideas in the chapter affect performance of real programs.

Fourth, in the interval since the second edition of this book, Moore’s law has marched onward so that we now have processors with 200 million transistors, DRAM chips with a billion transistors, and clock rates of multiple gigahertz. The “Real Stuff” examples have been updated to describe such chips. This edition also includes AMD64/IA-32e, the 64-bit address version of the long-lived 80x86 architecture, which appears to be the nemesis of the more recent IA-64. It also reflects the transition from parallel buses to serial networks and switches. Later chapters describe Google, which was born after the second edition, in terms of its cluster technology and in novel uses of search.

Fifth, although many computer science and engineering students enjoy information technology for technology’s sake, some have more altruistic interests. This latter group tends to have more women and underrepresented minorities. Consequently, we have added a new book element, “Computers in the Real World,” two-page layouts found between each chapter. Our perspective is that information technology is more valuable for humanity than most other topics you could study—whether it is preserving our art heritage, helping the Third World, saving our environment, or even changing political systems—and so we demonstrate our view with concrete examples of nontraditional applications. We think readers of these segments will have a greater appreciation of the computing culture beyond

the inherently interesting technology, much like those who read the history sections at the end of each chapter

Finally, books are like people: they usually get larger as they get older. By using technology, we have managed to do all the above and yet shrink the page count by hundreds of pages. As the table illustrates, the core portion of the book for hardware and software readers is on paper, but sections that some readers would value more than others are found on the companion CD. This technology also allows your authors to provide longer histories and more extensive exercises without concerns about lengthening the book. Once we added the CD to the book, we could then include a great deal of free software and tutorials that many instructors have told us they would like to use in their courses. This hybrid paper-CD publication weighs about 30% less than it did six years ago—an impressive goal for books as well as for people.

Instructor Support

We have collected a great deal of material to help instructors teach courses using this book. Solutions to exercises, figures from the book, lecture notes, lecture slides, and other materials are available to adopters from the publisher. Check the publisher's Web site for more information:

www.mkp.com/companions/1558606041

Concluding Remarks

If you read the following acknowledgments section, you will see that we went to great lengths to correct mistakes. Since a book goes through many printings, we have the opportunity to make even more corrections. If you uncover any remaining, resilient bugs, please contact the publisher by electronic mail at *cod3bugs@mkp.com* or by low-tech mail using the address found on the copyright page. The first person to report a technical error will be awarded a \$1.00 bounty upon its implementation in future printings of the book!

This book is truly collaborative, despite one of us running a major university. Together we brainstormed about the ideas and method of presentation, then individually wrote about one-half of the chapters and acted as reviewer for every draft of the other half. The page count suggests we again wrote almost exactly the same number of pages. Thus, we equally share the blame for what you are about to read.

Acknowledgments for the Third Edition

We'd like to again express our appreciation to **Jim Larus** for his willingness in contributing his expertise on assembly language programming, as well as for welcoming readers of this book to use the simulator he developed and maintains. Our

exercise editor **Dan Sorin** took on the Herculean task of adding new exercises and answers. **Peter Ashenden** worked similarly hard to collect and organize the companion CD.

We are grateful to the many instructors who answered the publisher's surveys, reviewed our proposals, and attended focus groups to analyze and respond to our plans for this edition. They include the following individuals: Michael Anderson (University of Hartford), David Bader (University of New Mexico), Rusty Baldwin (Air Force Institute of Technology), John Barr (Ithaca College), Jack Briner (Charleston Southern University), Mats Brorsson (KTH, Sweden), Colin Brown (Franklin University), Lori Carter (Point Loma Nazarene University), John Casey (Northeastern University), Gene Chase (Messiah College), George Cheney (University of Massachusetts, Lowell), Daniel Citron (Jerusalem College of Technology, Israel), Albert Cohen (INRIA, France), Lloyd Dickman (PathScale), Jose Duato (Universidad Politécnica de Valencia, Spain), Ben Dugan (University of Washington), Derek Eager (University of Saskatchewan, Canada), Magnus Ekman (Chalmers University of Technology, Sweden), Ata Elahi (Southern Connecticut State University), Soundararajan Ezekiel (Indiana University of Pennsylvania), Ernest Ferguson (Northwest Missouri State University), Michael Fry (Lebanon Valley College, Pennsylvania), R. Gaede (University of Arkansas at Little Rock), Jean-Luc Gaudiot (University of California, Irvine), Thomas Gendreau (University of Wisconsin, La Crosse), George Georgiou (California State University, San Bernardino), Paul Gillard (Memorial University of Newfoundland, Canada), Joe Grimes (California Polytechnic State University, SLO), Max Hailperin (Gustavus Adolphus College), Jayantha Herath (St. Cloud State University, Minnesota), Mark Hill (University of Wisconsin, Madison), Michael Hsaio (Virginia Tech), Richard Hughey (University of California, Santa Cruz), Tony Jebara (Columbia University), Elizabeth Johnson (Xavier University), Peter Kogge (University of Notre Dame), Morris Lancaster (BAH), Doug Lawrence (University of Montana), David Lilja (University of Minnesota), Nam Ling (Santa Clara University, California), Paul Lum (Agilent Technologies), Stephen Mann (University of Waterloo, Canada), Diana Marculescu (Carnegie Mellon University), Margaret McMahon (U.S. Naval Academy Computer Science), Uwe Meyer-Baese (Florida State University), Chris Milner (University of Virginia), Tom Pittman (Southwest Baptist University), Jalel Rejeb (San Jose State University, California), Bill Siever (University of Missouri, Rolla), Kevin Skadron (University of Virginia), Pam Smallwood (Regis University, Colorado), K. Stuart Smith (Rocky Mountain College), William J. Taffe (Plymouth State University), Michael E. Thomodakis (Texas A&M University), Ruppa K. Thulasiram (University of Manitoba, Canada), Ye Tung (University of South Alabama), Steve VanderLeest (Calvin College), Neal R. Wagner (University of Texas at San Antonio), and Kent Wilken (University of California, Davis).

We are grateful too to those who carefully read our draft manuscripts; some read successive drafts to help ensure new errors didn't creep in as we revised. They include Krste Asanovic (Massachusetts Institute of Technology), Jean-Loup Baer (University of Washington), David Brooks (Harvard University), Doug Clark (Princeton University), Dan Connors (University of Colorado at Boulder), Matt Farrens (University of California, Davis), Manoj Franklin (University of Maryland College Park), John Greiner (Rice University), David Harris (Harvey Mudd College), Paul Hilfinger (University of California, Berkeley), Norm Jouppi (Hewlett-Packard), David Kaeli (Northeastern University), David Oppenheimer (University of California, Berkeley), Timothy Pinkston (University of Southern California), Mark Smotherman (Clemson University), and David Wood (University of Wisconsin, Madison).

To help us meet our goal of creating 70% new exercises and solutions for this edition, we recruited several graduate students recommended to us by their professors. We are grateful for their creativity and persistence: Michael Black (University of Maryland), Lei Chen (University of Rochester), Nirav Dave (Massachusetts Institute of Technology), Wael El Essawy (University of Rochester), Nikil Mehta (Brown University), Nicholas Nelson (University of Rochester), Aaron Smith (University of Texas, Austin), and Charlie Wang (Duke University).

We would like to especially thank **Mark Smotherman** for making a careful final pass to find technical and writing glitches that significantly improved the quality of this edition.

We wish to thank the extended Morgan Kaufmann family for agreeing to publish this book again under the able leadership of **Denise Penrose**. She developed the vision of the hybrid paper-CD book and recruited the many people above who played important roles in developing the book.

Simon Crump managed the book production process, and **Summer Block** coordinated the surveying of users and their responses. We thank also the many freelance vendors who contributed to this volume, especially **Nancy Logan** and **Dartmouth Publishing, Inc.**, our compositors.

The contributions of the nearly 100 people we mentioned here have made this third edition our best book yet. Enjoy!

David A. Patterson


John L. Hennessy

1

Computer Abstractions and Technology

*Civilization advances by extending
the number of important operations
which we can perform without
thinking about them.*

Alfred North Whitehead
An Introduction to Mathematics, 1911

1.1	Introduction	3
1.2	Below Your Program	11
1.3	Under the Covers	15
1.4	Real Stuff: Manufacturing Pentium 4 Chips	28
1.5	Fallacies and Pitfalls	33
1.6	Concluding Remarks	35
 1.7	Historical Perspective and Further Reading	36
1.8	Exercises	36

1.1

Introduction

Welcome to this book! We're delighted to have this opportunity to convey the excitement of the world of computer systems. This is not a dry and dreary field, where progress is glacial and where new ideas atrophy from neglect. No! Computers are the product of the incredibly vibrant information technology industry, all aspects of which are responsible for almost 10% of the gross national product of the United States. This unusual industry embraces innovation at a breathtaking rate. Since 1985 there have been a number of new computers whose introduction appeared to revolutionize the computing industry; these revolutions were cut short only because someone else built an even better computer.

This race to innovate has led to unprecedented progress since the inception of electronic computing in the late 1940s. Had the transportation industry kept pace with the computer industry, for example, today we could travel coast to coast in about a second for roughly a few cents. Take just a moment to contemplate how such an improvement would change society—living in Tahiti while working in San Francisco, going to Moscow for an evening at the Bolshoi Ballet—and you can appreciate the implications of such a change.

Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and the industrial revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected our everyday lives profoundly and also changed the ways in which the search for new knowledge is carried out. There is now a new vein of scientific investigation, with computational scientists joining theoretical and experimental scientists in the exploration of new frontiers in astronomy, biology, chemistry, physics, . . .

The computer revolution continues. Each time the cost of computing improves by another factor of 10, the opportunities for computers multiply. Applications that were economically infeasible suddenly become practical. In the recent past, the following applications were "computer science fiction."

- *Automatic teller machines:* A computer placed in the wall of banks to distribute and collect cash would have been a ridiculous concept in the 1950s, when the cheapest computer cost at least \$500,000 and was the size of a car.
- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce pollution and improve fuel efficiency via engine controls and increase safety through the prevention of dangerous skids and through the inflation of air bags to protect occupants in a crash.
- *Laptop computers:* Who would have dreamed that advances in computer systems would lead to laptop computers, allowing students to bring computers to coffeehouses and on airplanes?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences is hundreds of millions of dollars. It's unlikely that anyone would have considered this project had the computer costs been 10 to 100 times higher, as they would have been 10 to 20 years ago.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the World Wide Web has transformed our society. Among its uses are distributing news, sending flowers, buying from online catalogues, taking electronic tours to help pick vacation spots, finding others who share your esoteric interests, and even more mundane topics like finding the lecture notes of the authors of your textbooks.

Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, and explain why computers are omnipresent. Tomorrow's science fiction computer applications are the cashless society, automated intelligent highways, and genuinely ubiquitous computing: no one carries computers because they are available everywhere.

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (discussed in Sections 1.3 and 1.4) is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have different design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three different classes of applications.

Desktop computers are possibly the best-known form of computing and are characterized by the personal computer, which most readers of this book have probably used extensively. Desktop computers emphasize delivering good performance to a single user at low cost and usually are used to execute third-party software, also called shrink-wrap software. Desktop computing is one of the largest markets for computers, and the evolution of many computing technologies is driven by this class of computing, which is only about 30 years old!

Servers are the modern form of what was once mainframes, minicomputers, and supercomputers, and are usually accessed only via a network. Servers are oriented to carrying large workloads, which may consist of either single complex applications, usually a scientific or engineering application, or handling many small jobs, such as would occur in building a large Web server. These applications are often based on software from another source (such as a database or simulation system), but are often modified or customized for a particular function. Servers are built from the same basic technology as desktop computers, but provide for greater expandability of both computing and input/output capacity. As we will see in the Chapter 4, the performance of a server can be measured in several different ways, depending on the application of interest. In general, servers also place a greater emphasis on dependability, since a crash is usually more costly than it would be on a single-user desktop computer.

Servers span the widest range in cost and capability. At the low end, a server may be little more than a desktop machine without a screen or keyboard and with a cost of a thousand dollars. These low-end servers are typically used for file storage, small business applications, or simple web serving. At the other extreme are **supercomputers**, which at the present consist of hundreds to thousands of processors, and usually gigabytes to **terabytes** of memory and terabytes to petabytes of storage, and cost millions to hundreds of millions of dollars. Supercomputers are usually used for high-end scientific and engineering calculations, such as weather forecasting, oil exploration, protein structure determination, and other large-scale problems. Although such supercomputers represent the peak of computing capability, they are a relatively small fraction of the servers and a relatively small fraction of the overall computer market in terms of total revenue.

Embedded computers are the largest class of computers and span the widest range of applications and performance. Embedded computers include the microprocessors found in your washing machine and car, the computers in a cell phone

desktop computer A computer designed for use by an individual, usually incorporating a graphics display, keyboard, and mouse.

Server A computer used for running larger programs for multiple users often simultaneously and typically accessed only via a network.

supercomputer A class of computers with the highest performance and cost; they are configured as servers and typically cost millions of dollars.

terabyte Originally 1,099,511,627,776 (2^{40}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000,000 (10^{12}) bytes.

embedded computer A computer inside another device used for running one predetermined application or collection of software.

or personal digital assistant, the computers in a video game or digital television, and the networks of processors that control a modern airplane or cargo ship. Embedded computing systems are designed to run one application or one set of related applications, which is normally integrated with the hardware and delivered as a single system; thus, despite the large number of embedded computers, most users never really see that they are using a computer!

Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a cell phone: the processor need only be as fast as necessary to handle its limited function, and beyond that, minimizing cost and power are the most important objectives. Despite their low cost, embedded computers often have the least tolerance for failure, since the results can vary from upsetting (when your new television crashes) to devastating (such as might occur when the computer in a plane or car crashes). In consumer-oriented embedded applications, such as a digital home appliance, dependability is achieved primarily through simplicity—the emphasis is on doing one function, as perfectly as possible. In large embedded systems, techniques of redundancy, which are used in servers, are often employed. Although this book focuses on general-purpose computers, most of the concepts apply directly, or with slight modifications, to embedded computers. In several places, we will touch on some of the unique aspects of embedded computers.

Figure 1.1 shows that during the last several years, the growth in the number of embedded computers has been much faster (40% compounded annual growth rate) than the growth rate among desktop computers and servers (9% annually). Note that the embedded computers include cell phones, video games, digital TVs and set-top boxes, personal digital assistants, and a variety of such consumer devices. Note that this data does not include low-end embedded control devices that use 8-bit and 16-bit processors.

Elaboration: Elaborations are short sections used throughout the text to provide more detail on a particular subject, which may be of interest. Disinterested readers may skip over an elaboration, since the subsequent material will never depend on the contents of the elaboration.

Many embedded processors are designed using *processor cores*, a version of a processor written in a hardware description language such as Verilog or VHDL. The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip. The availability of synthesis tools that can generate a chip from a Verilog specification, together with the capacity of modern silicon chips, has made such special-purpose processors highly attractive. Since the core can be synthesized for different semiconductor manufacturing lines, using a core provides flexibility in choosing a manufacturer as well. In the last few years, the use of cores has

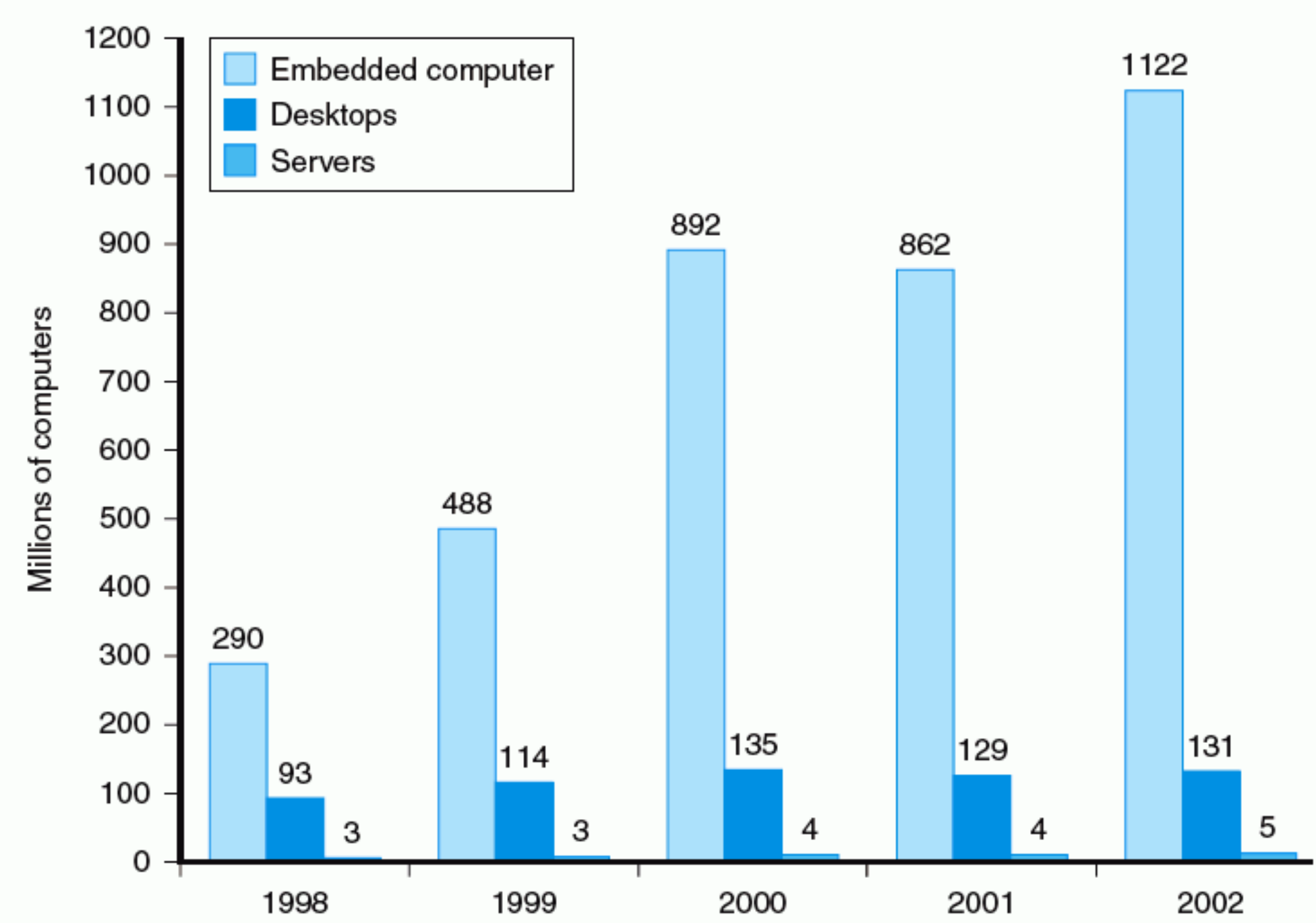


FIGURE 1.1 The number of distinct processors sold between 1998 and 2002. These counts are obtained somewhat differently, so some caution is required in interpreting the results. For example, the totals for desktops and servers count complete computer systems, because some fraction of these include multiple processors, the number of processors sold is somewhat higher, but probably by only 10–20% in total (since the servers, which may average more than one processor per system, are only about 3% of the desktop sales, which are predominantly single-processor systems). The totals for embedded computers actually count processors, many of which are not even visible, and in some cases there may be multiple processors per device.

been growing very fast. For example, in 1998 only 31% of the embedded processors were cores. By 2002, 56% of the embedded processors were cores. Furthermore, while the overall growth rate in the embedded market has been 40% per year, this growth has been primarily driven by cores, where the compounded annual growth rate has been 63%!

Figure 1.2 shows the major architectures sold in these markets with counts for each architecture, across all three types of products (embedded, desktop, and server). Only 32-bit and 64-bit processors are included, although 32-bit processors are the vast majority for most of the architectures.

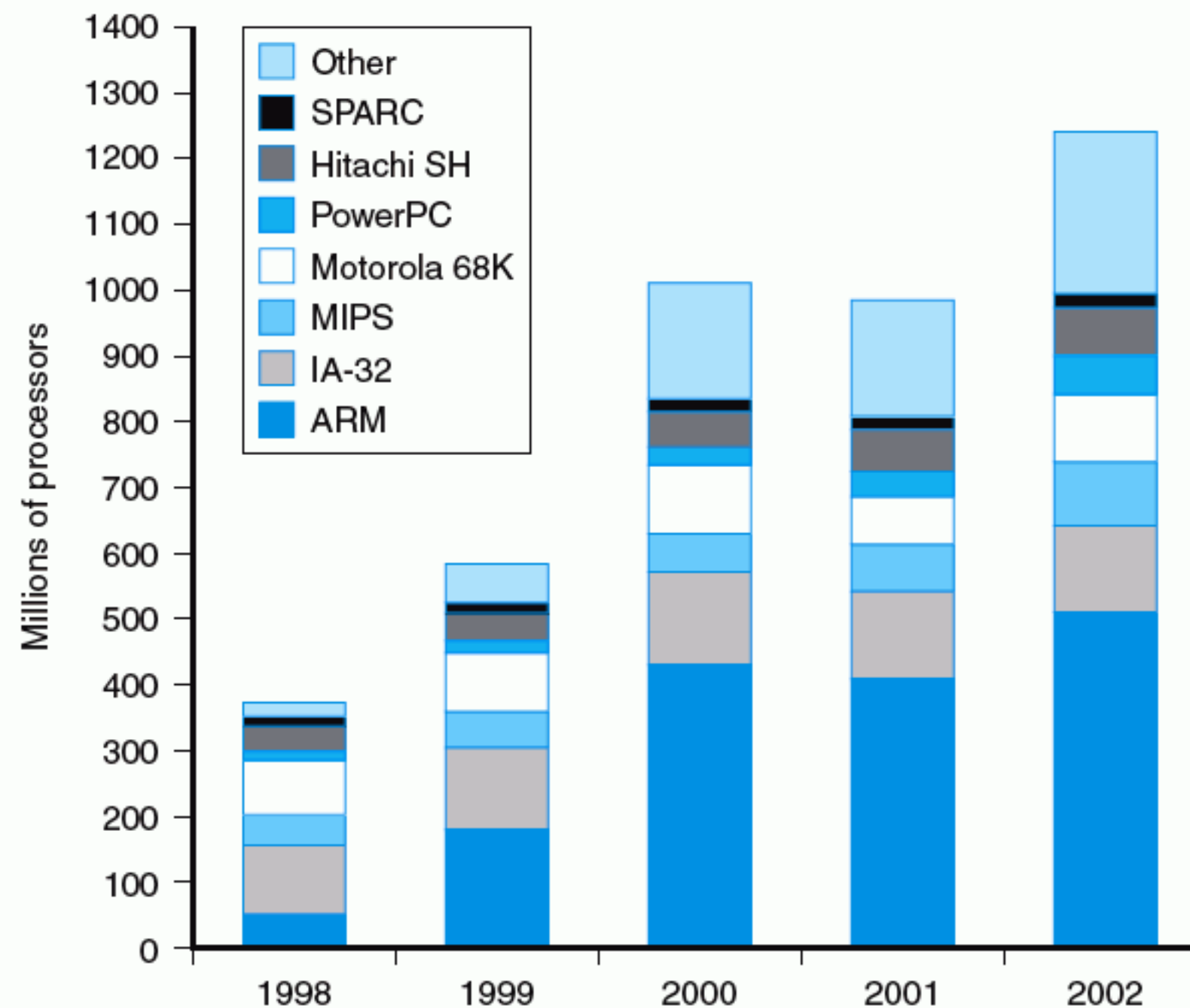


FIGURE 1.2 Sales of microprocessors between 1998 and 2002 by instruction set architecture combining all uses. The “other” category refers to processors that are either application-specific or customized architectures. In the case of ARM, roughly 80% of the sales are for cell phones, where an ARM core is used in conjunction with application-specific logic on a chip.

What You Can Learn in This Book

Successful programmers have always been concerned about the performance of their programs because getting results to the user quickly is critical in creating successful software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer’s memory. Thus programmers often followed a simple credo: Minimize memory space to make programs fast. In the last decade, advances in computer design and memory technology have greatly reduced the importance of small memory size in most applications other than those in embedded computing systems.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the hierarchical nature of memories and the parallel nature of processors. Programmers who seek to build competitive versions of compilers, operating systems, databases, and even applications will therefore need to increase their knowledge of computer organization.

We are honored to have the opportunity to explain what's inside this revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer. By the time you complete this book, we believe you will be able to answer the following questions:

- How are programs written in a high-level language, such as C or Java, translated into the language of the hardware, and how does the hardware execute the resulting program? Comprehending these concepts forms the basis of understanding the aspects of both the hardware and software that affect program performance.
- What is the interface between the software and the hardware, and how does software instruct the hardware to perform needed functions? These concepts are vital to understanding how to write many kinds of software.
- What determines the performance of a program, and how can a programmer improve the performance? As we will see, this depends on the original program, the software translation of that program into the computer's language, and the effectiveness of the hardware in executing the program.
- What techniques can be used by hardware designers to improve performance? This book will introduce the basic concepts of modern computer design. The interested reader will find much more material on this topic in our advanced book, *A Computer Architecture: A Quantitative Approach*.

Without understanding the answers to these questions, improving the performance of your program on a modern computer, or evaluating what features might make one computer better than another for a particular application, will be a complex process of trial and error, rather than a scientific procedure driven by insight and analysis.

This first chapter lays the foundation for the rest of the book. It introduces the basic ideas and definitions, places the major components of software and hardware in perspective, and introduces integrated circuits, the technology that fuels the computer revolution. In this chapter, and later ones, you will likely see a lot of new words, or words that you may have heard, but are not sure what they mean. Don't panic! Yes, there is a lot of special terminology used in describing modern computers, but the terminology actually helps since it enables us to describe precisely a function or capability. In addition, computer designers (including your authors) *love* using **acronyms**, which are *easy* to understand once you know what the letters stand for! To help you remember and locate terms, we have included a highlighted definition of every term, the first time it appears in the text. After a short time of working with the terminology, you will be fluent, and your friends

acronym A word constructed by taking the initial letters of string of words. For example: RAM is an acronym for Random Access Memory, and CPU is an acronym for Central Processing Unit.

will be impressed as you correctly use words such as BIOS, DIMM, CPU, cache, DRAM, ATA, PCI, and many others.

To reinforce how the software and hardware systems used to run a program will affect performance, we use a special section, “Understanding Program Performance,” throughout the book, with the first one appearing below. These elements summarize important insights into program performance.

Understanding
Program
Performance

The performance of a program depends on a combination of the effectiveness of the algorithms used in the program, the software systems used to create and translate the program into machine instructions, and the effectiveness of the computer in executing those instructions, which may include I/O operations. The following table summarizes how the hardware and software affect performance.

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of machine instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 5, 6, and 7
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapter 8

Check
Yourself

“Check Yourself” sections are designed to help readers assess whether they have comprehended the major concepts introduced in a chapter and understand the implications of those concepts. Some “Check Yourself” questions have simple answers; others are for discussion among a group. Answers to the specific questions can be found at the end of the chapter. “Check Yourself” questions appear only at the end of a section, making it easy to skip them if you are sure you understand the material.

1. Section 1.1 showed that the number of embedded processors sold every year greatly outnumbers the number of desktop processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of desktop computers in your home?

2. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?
 - The algorithm chosen
 - The programming language or compiler
 - The operating system
 - The processor
 - The I/O system and devices

1.2

Below Your Program

A typical application, such as a word processor or a large database system, may consist of hundreds of thousands to millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. As we will see, the hardware in a computer can only execute extremely simple low-level instructions. To go from a complex application to the simple instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions.

These layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of **systems software** sitting between the hardware and applications software, as shown in Figure 1.3.

There are many types of systems software, but two types of systems software are central to every computer system today: an operating system and a compiler. An **operating system** interfaces between a user's program and the hardware and provides a variety of services and supervisory functions. Among the most important functions are

- handling basic input and output operations
- allocating storage and memory
- providing for sharing the computer among multiple applications using it simultaneously

Examples of operating systems in use today are Windows, Linux, and MacOS.

Compilers perform another vital function: the translation of a program written in a high-level language, such as C or Java, into instructions that the hardware

In Paris they simply stared when I spoke to them in French; I never did succeed in making those idiots understand their own language.

Mark Twain, *The Innocents Abroad*, 1869

systems software Software that provides services that are commonly useful, including operating systems, compilers, and assemblers.

operating system Supervising program that manages the resources of a computer for the benefit of the programs that run on that machine.

compiler A program that translates high-level language statements into assembly language statements.

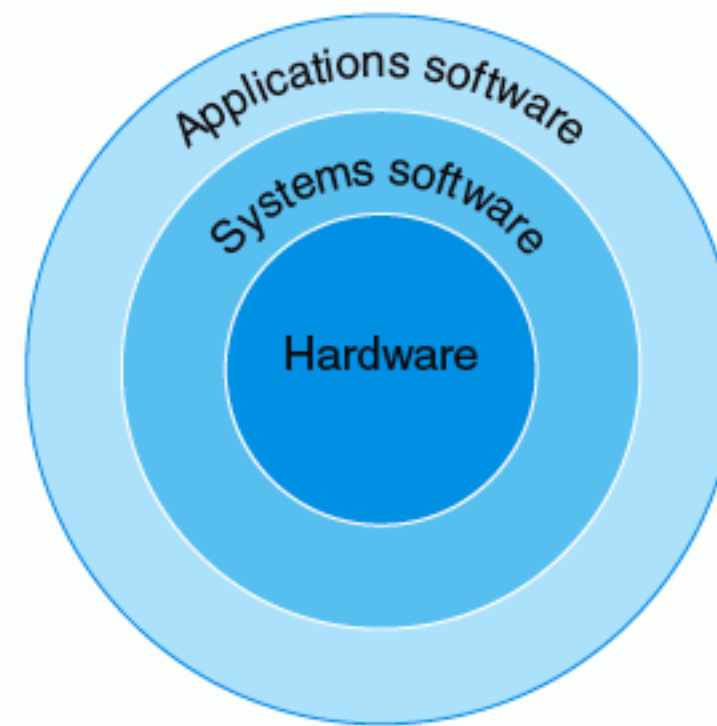


FIGURE 1.3 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.

can execute. Given the sophistication of modern programming languages and the simple instructions executed by the hardware, the translation from a high-level language program to hardware instructions is complex. We will give a brief overview of the process and return to the subject in Chapter 2.

From a High-Level Language to the Language of Hardware

To actually speak to an electronic machine, you need to send electrical signals. The easiest signals for machines to understand are *on* and *off*, and so the machine alphabet is just two letters. Just as the 26 letters of the English alphabet do not limit how much can be written, the two letters of the computer alphabet do not limit what computers can do. The two symbols for these two letters are the numbers 0 and 1, and we commonly think of the machine language as numbers in base 2, or *binary numbers*. We refer to each “letter” as a **binary digit** or **bit**. Computers are slaves to our commands, which are called instructions. Instructions, which are just collections of bits that the computer understands, can be thought of as numbers. For example, the bits

1000110010100000

tell one computer to add two numbers. Chapter 3 explains why we use numbers for instructions *and* data; we don’t want to steal that chapter’s thunder, but using numbers for both instructions and data is a foundation of computing.

binary digit Also called a bit. One of the two numbers in base 2 (0 or 1) that are the components of information.

The first programmers communicated to computers in binary numbers, but this was so tedious that they quickly invented new notations that were closer to the way humans think. At first these notations were translated to binary by hand, but this process was still tiresome. Using the machine to help program the machine, the pioneers invented programs to translate from symbolic notation to binary. The first of these programs was named an **assembler**. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

```
add A,B
```

and the assembler would translate this notation into

```
1000110010100000
```

This instruction tells the computer to add the two numbers A and B. The name coined for this symbolic language, still used today, is **assembly language**.

Although a tremendous improvement, assembly language is still far from the notation a scientist might like to use to simulate fluid flow or that an accountant might use to balance the books. Assembly language requires the programmer to write one line for every instruction that the machine will follow, forcing the programmer to think like the machine.

The recognition that a program could be written to translate a more powerful language into computer instructions was one of the great breakthroughs in the early days of computing. Programmers today owe their productivity—and their sanity—to the creation of **high-level programming languages** and compilers that translate programs in such languages into instructions.

A compiler enables a programmer to write this high-level language expression:

```
A + B
```

The compiler would compile it into this assembly language statement:

```
add A,B
```

The assembler would translate this statement into the binary instruction that tells the computer to add the two numbers A and B:

```
1000110010100000
```

Figure 1.4 shows the relationships among these programs and languages.

High-level programming languages offer several important benefits. First, they allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols (see Figure 1.4). Moreover, they allow languages to

assembler A program that translates a symbolic version of instructions into the binary version.

assembly language A symbolic representation of machine instructions.

high-level programming language A portable language such as C, Fortran, or Java composed of words and algebraic notation that can be translated by a compiler into assembly language.

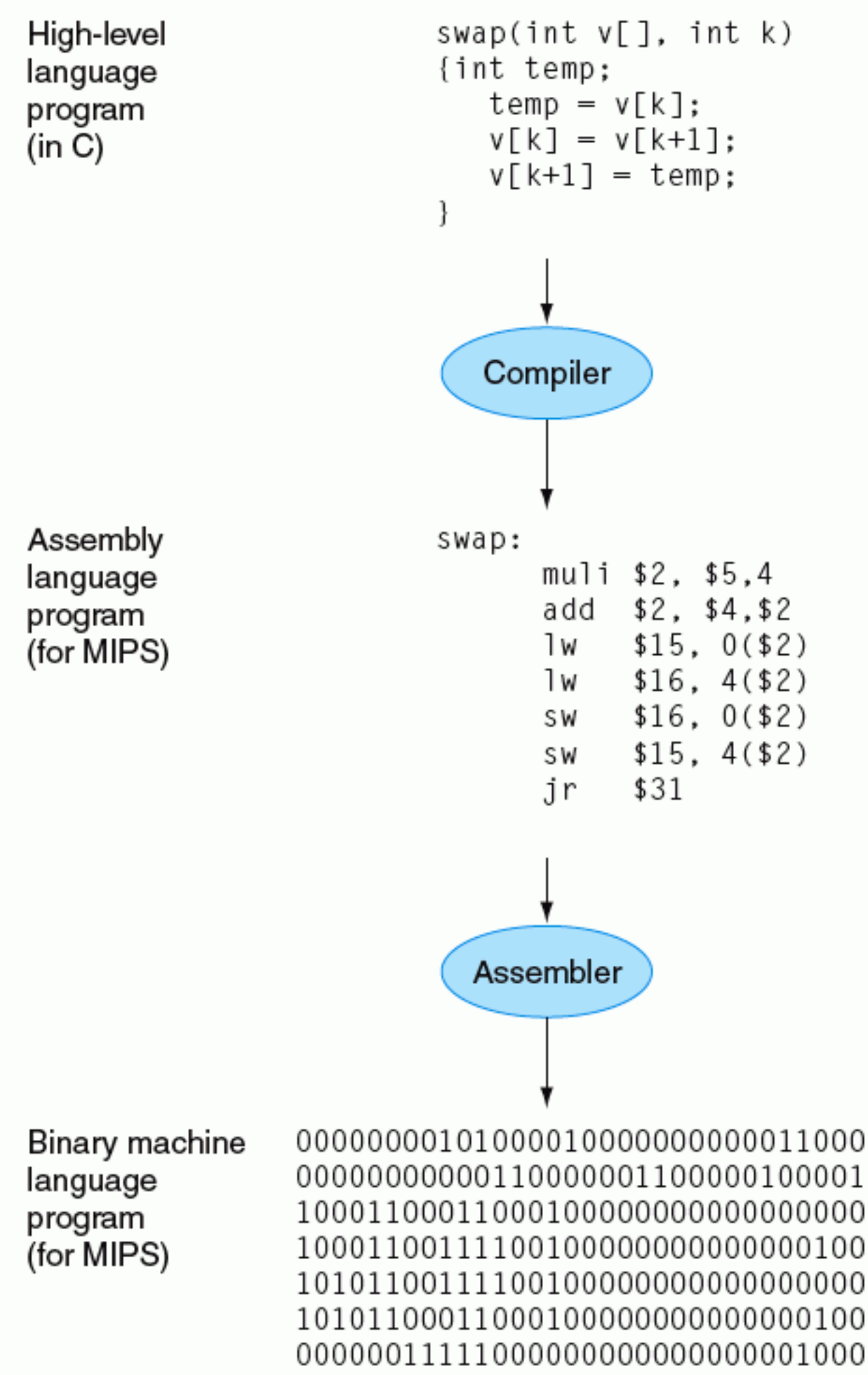


FIGURE 1.4 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in Chapter 2.

be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on.

The second advantage of programming languages is improved programmer productivity. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea. Conciseness is a clear advantage of high-level languages over assembly language.

The final advantage is that programming languages allow programs to be independent of the computer on which they were developed, since compilers and assemblers can translate high-level language programs to the binary instructions of any machine. These three advantages are so strong that today little programming is done in assembly language.

1.3 Under the Covers

Now that we have looked below your program to uncover the underlying software, let's open the covers of the computer to learn about the underlying hardware. The underlying hardware in any computer performs the same basic functions: inputting data, outputting data, processing data, and storing data. How these functions are performed is the primary topic of this book, and subsequent chapters deal with different parts of these four tasks. When we come to an important point in this book, a point so important that we hope you will remember it forever, we emphasize it by identifying it as a "Big Picture" item. We have about a dozen Big Pictures in this book, with the first being the five components of a computer that perform the tasks of inputting, outputting, processing, and storing data.

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. Figure 1.5 shows the standard organization of a computer. This organization is independent of hardware technology: You can place every piece of every computer, past and present, into one of these five categories. To help you keep all this in perspective, the five components of a computer are shown on the front page of the following chapters, with the portion of interest to that chapter highlighted.

Figure 1.6 shows a typical desktop computer with keyboard, mouse, screen, and a box containing even more hardware. What is not visible in the photograph is a network that connects the computer to other computers. This photograph reveals two of the key components of computers: **input devices**, such as the keyboard and mouse, and **output devices**, such as the screen. As the names suggest, input feeds the computer, and output is the result of computation sent to the user. Some devices, such as networks and disks, provide both input and output to the computer.

The BIG Picture

input device A mechanism through which the computer is fed information, such as the keyboard or mouse.

output device A mechanism that conveys the result of a computation to a user or another computer.

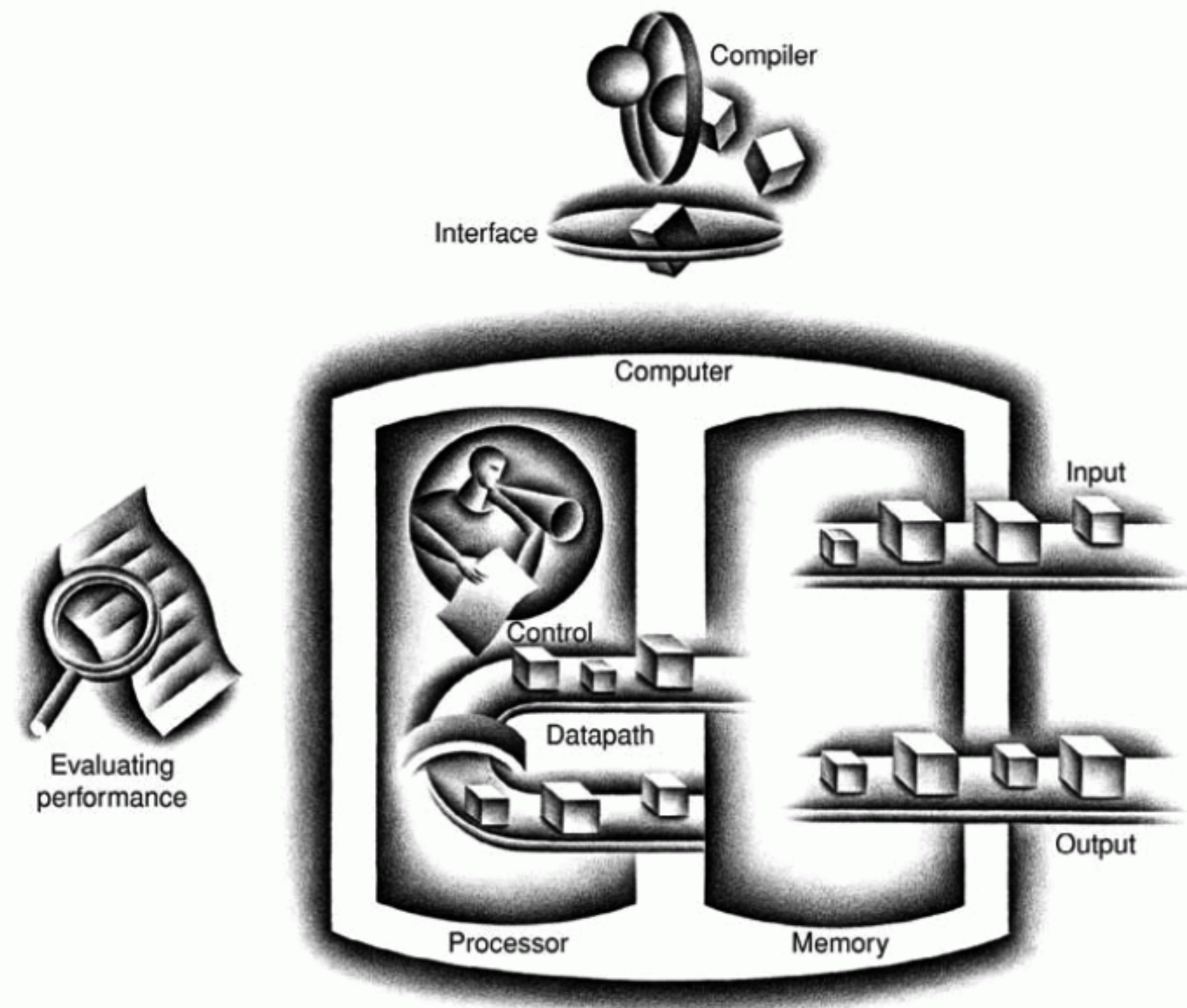


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Chapter 8 describes input/output (I/O) devices in more detail, but let's take an introductory tour through the computer hardware, starting with the external I/O devices.

I got the idea for the mouse while attending a talk at a computer conference. The speaker was so boring that I started daydreaming and hit upon the idea.

Doug Engelbart

Anatomy of a Mouse

Although many users now take mice for granted, the idea of a pointing device such as a mouse was first shown by Engelbart using a research prototype in 1967. The Alto, which was the inspiration for all workstations as well as for the Macintosh, included a mouse as its pointing device in 1973. By the 1990s, all desktop computers included this device, and new user interfaces based on graphics displays and mice became the norm.



FIGURE 1.6 A desktop computer. The liquid crystal display (LCD) screen is the primary output device, and the keyboard and mouse are the primary input devices. The box contains the processor as well as additional I/O devices. This system is a Dell Optiplex GX260.

The original mouse was electromechanical and used a large ball that when rolled across a surface would cause an x and y counter to be incremented. The amount of increase in each counter told how far the mouse had been moved.

The electromechanical mouse has largely been replaced by the newer all-optical mouse. The optical mouse is actually a miniature optical processor including an LED to provide lighting, a tiny black-and-white camera, and a simple optical processor. The LED illuminates the surface underneath the mouse; the camera takes 1500 sample pictures a second under the illumination. Successive pictures are sent to a simple optical processor that compares the images and determines whether the mouse has moved and how far. The replacement of the electromechanical mouse by the electro-optical mouse is an illustration of a common phenomenon where the decreasing costs and higher reliability of electronics cause an electronic solution to replace the older electromechanical technology.

Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light and watched a computer reveal its innermost workings.

Ivan Sutherland, the “father” of computer graphics, quoted in “Computer Software for Graphics,” *Scientific American*, 1984

cathode ray tube (CRT) display A display, such as a television set, that displays an image using an electron beam scanned across a screen.

pixel The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

flat panel display, liquid crystal display A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

active matrix display A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

Through the Looking Glass

The most fascinating I/O device is probably the graphics display. Based on television technology, a **cathode ray tube (CRT) display** scans an image one line at a time, 30 to 75 times per second. At this *refresh rate*, people don’t notice a flicker on the screen.

The image is composed of a matrix of picture elements, or **pixels**, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix ranges in size from 512×340 to 1920×1280 pixels in 2003. The simplest display has 1 bit per pixel, allowing it to be black or white. For displays that support 256 different shades of black and white, sometimes called *gray-scale* displays, 8 bits per pixel are required. A color display might use 8 bits for each of the three colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

All laptop and handheld computers, calculators, cellular phones, and many desktop computers use **flat-panel** or **liquid crystal displays (LCDs)** instead of CRTs to get a thin, low-power display. The main difference is that the LCD pixel is not the source of light; instead it controls the transmission of light. A typical LCD includes rod-shaped molecules in a liquid that form a twisting helix that bends light entering the display, from either a light source behind the display or less often from reflected light. The rods straighten out when a current is applied and no longer bend the light; since the liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent. Today, most LCD displays use an **active matrix** that has a tiny transistor switch at each pixel to precisely control current and make sharper images. As in a CRT, a red-green-blue mask associated with each pixel determines the intensity of the three color components in the final image; in a color active matrix LCD, there are three transistor switches at each pixel.

No matter what the display, the computer hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented on-screen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 1.7 shows a frame buffer with 4 bits per pixel.

The goal of the bit map is to faithfully represent what is on the screen. The challenges in graphics systems arise because the human eye is very good at detecting even subtle changes on the screen. For example, when the screen is being updated, the eye can detect the inconsistency between the portion of the screen that has changed and that which hasn’t.

Opening the Box

If we open the box containing the computer, we see a fascinating board of thin green plastic, covered with dozens of small gray or black rectangles. Figure 1.8

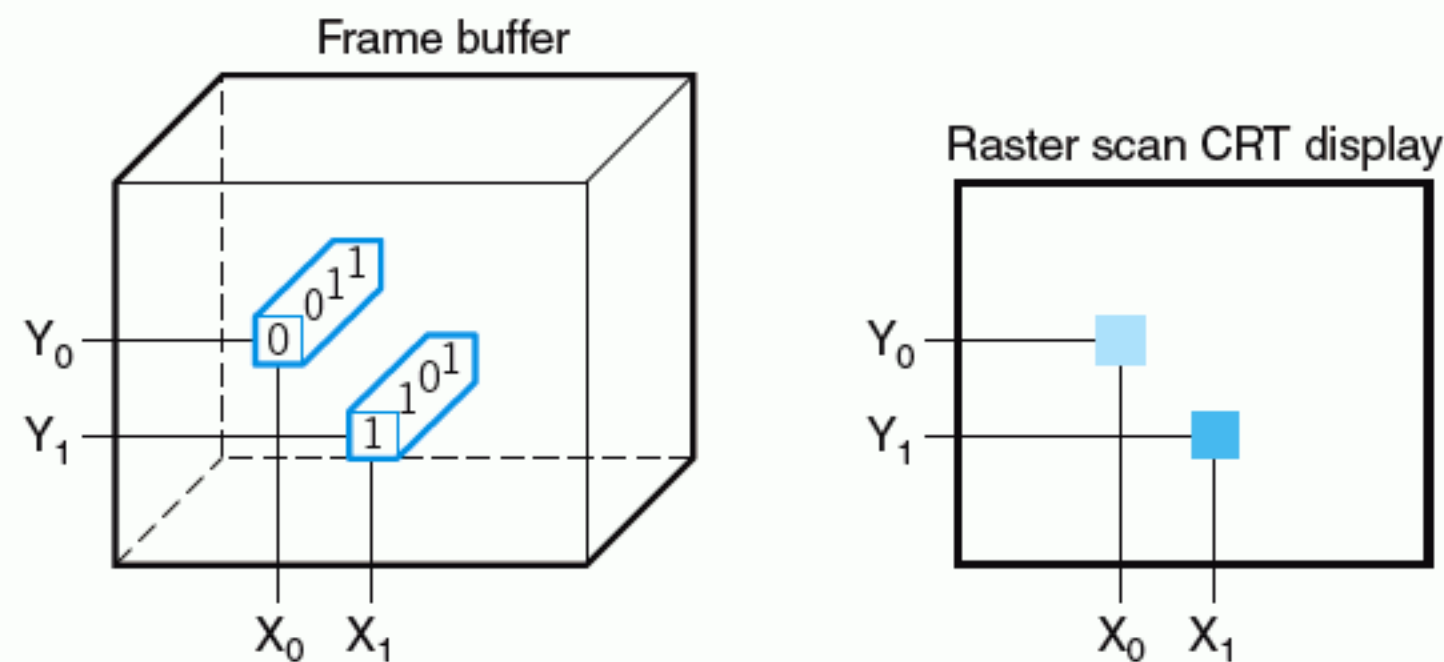


FIGURE 1.7 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade of gray on the screen than the bit pattern 1101 in pixel (X_1, Y_1) .

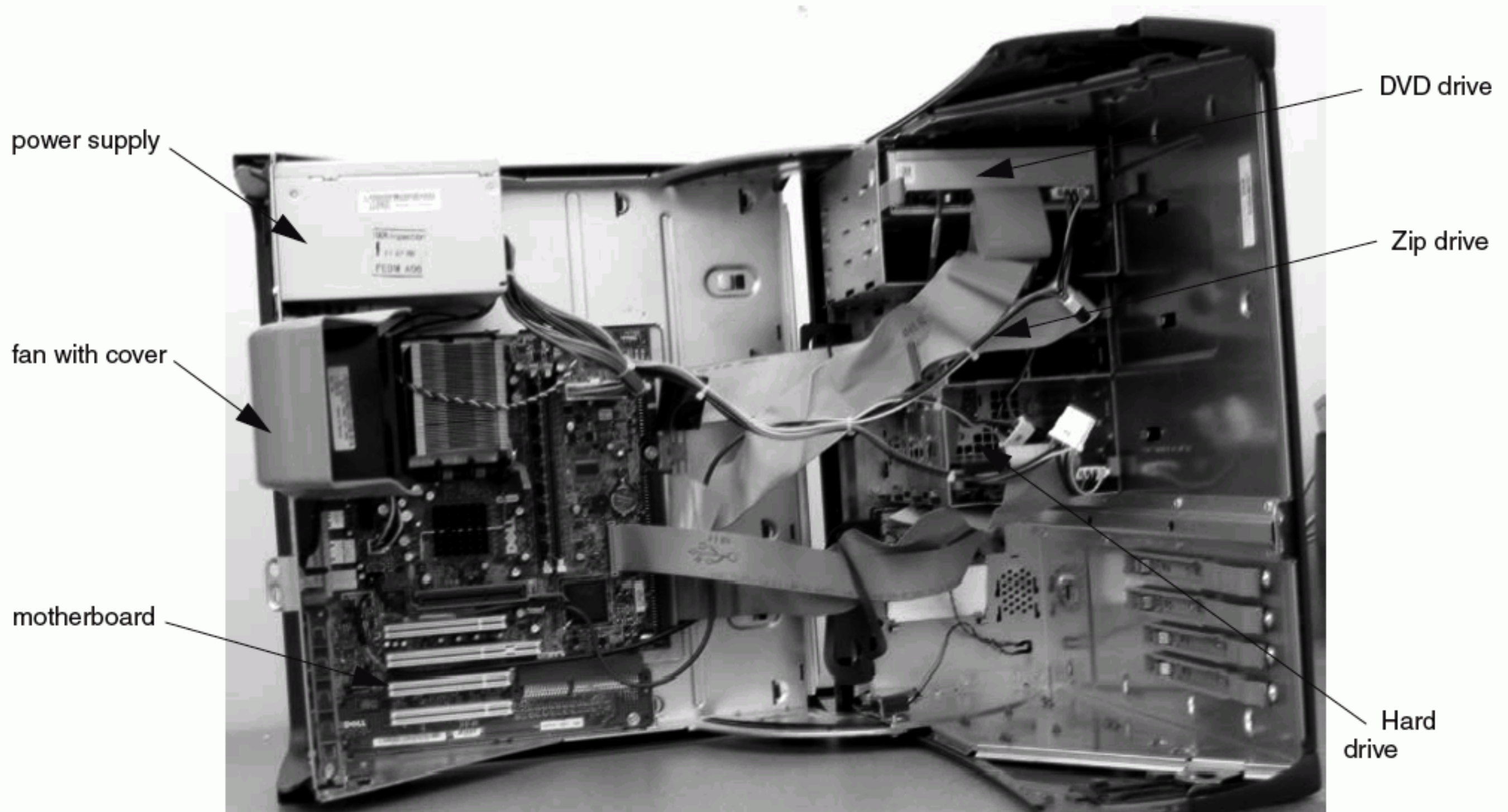


FIGURE 1.8 Inside the personal computer of Figure 1.6 on page 17. This packaging is sometimes called a clamshell because of the way it opens with hinges on one side. To see what's inside, let's start on the top left-hand side. The shiny metal box on the top far left side is the power supply. Just below that on the far left is the fan, with its cover pulled back. To the right and below the fan is a printed circuit board (PC board), called the *motherboard* in a PC, that contains most of the electronics of the computer; Figure 1.10 is a close-up of that board. The processor is the large raised rectangle just to the right of the fan. On the right side we see the bays designed to hold types of disk drives. The top bay contains a DVD drive, the middle bay a Zip drive, and the bottom bay contains a hard disk.

motherboard A plastic board containing packages of integrated circuits or chips, including processor, cache, memory, and connectors for I/O devices such as networks and disks.

integrated circuit Also called **chip**. A device combining dozens to millions of transistors.

memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

central processor unit (CPU) Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

datapath The component of the processor that performs arithmetic operations.

control The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

dynamic random access memory (DRAM) Memory built as an integrated circuit, it provides random access to any location.

cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

shows the contents of the desktop computer in Figure 1.6. This **motherboard** is shown vertically on the left with the power supply. Three disk drives—a DVD drive, Zip drive, and hard drive—appear on the right.

The small rectangles on the motherboard contain the devices that drive our advancing technology, **integrated circuits** or **chips**. The board is composed of three pieces: the piece connecting to the I/O devices mentioned earlier, the memory, and the processor. The I/O devices are connected via the two large boards attached perpendicularly to the motherboard toward the middle on the right-hand side.

The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. In Figure 1.8, memory is found on the two small boards that are attached perpendicularly toward the middle of the motherboard. Each small memory board contains eight integrated circuits.

The *processor* is the active part of the board, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. The processor is the large square below the memory boards in the lower-right corner of Figure 1.8. Occasionally, people call the processor the *CPU*, for the more bureaucratic-sounding **central processor unit**.

Descending even lower into the hardware, Figure 1.9 reveals details of the processor in Figure 1.8. The processor comprises two main components: datapath and control, the respective brawn and brain of the processor. The **datapath** performs the arithmetic operations, and **control** tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program. Chapter 5 explains the datapath and control for a straightforward implementation, and Chapter 6 describes the changes needed for a higher-performance design.

Descending into the depths of any component of the hardware reveals insights into the machine. The memory in Figure 1.10 is built from DRAM chips. *DRAM* stands for **dynamic random access memory**. Several DRAMs are used together to contain the instructions and data of a program. In contrast to sequential access memories such as magnetic tapes, the *RAM* portion of the term DRAM means that memory accesses take the same amount of time no matter what portion of the memory is read. Inside the processor is another type of memory—cache memory. **Cache memory** consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, static random access memory (SRAM). SRAM is faster but less dense, and hence more expensive, than DRAM.

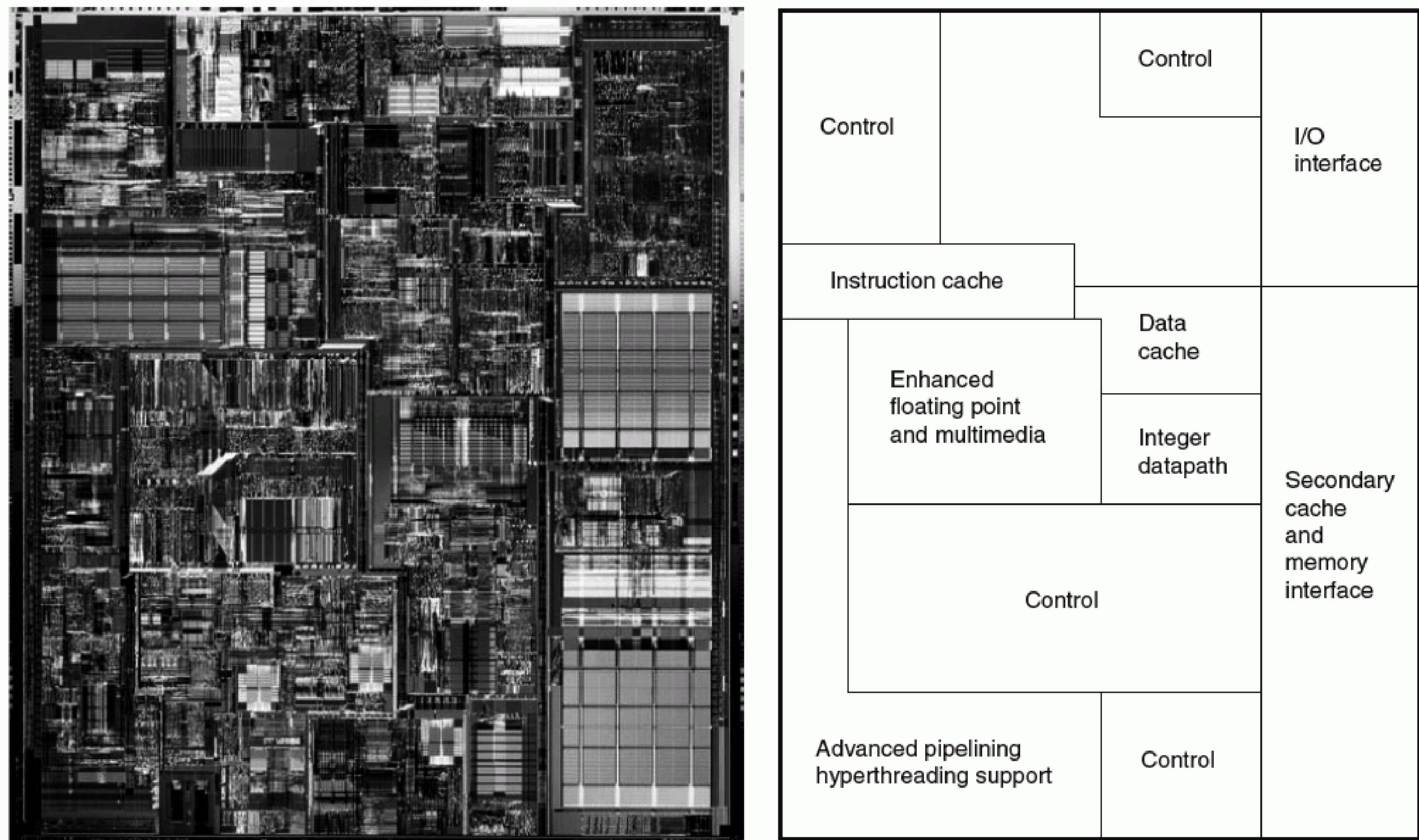
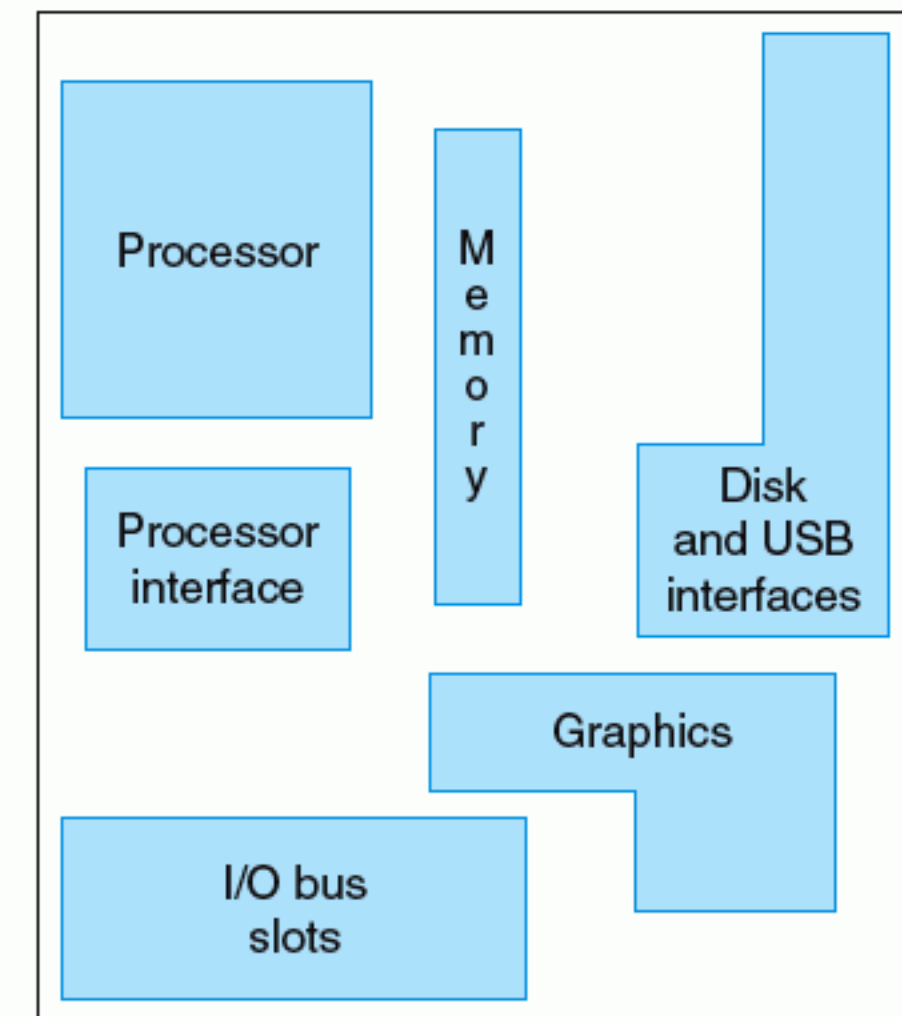
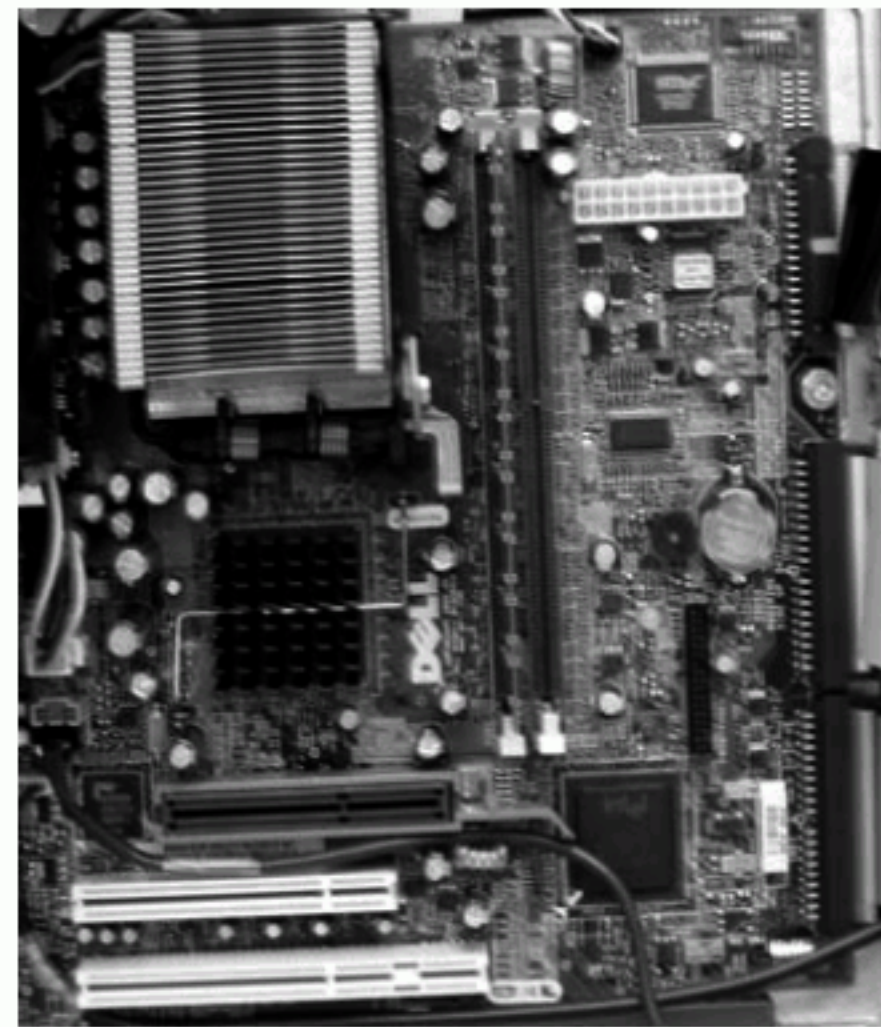


FIGURE 1.9 Inside the processor chip used on the board shown in Figure 1.8. The left-hand side is a microphotograph of the Pentium 4 processor chip, and the right-hand side shows the major blocks in the processor.

You may have noticed a common theme in both the software and the hardware descriptions: delving into the depths of hardware or software reveals more information or, conversely, lower-level details are hidden to offer a simpler model at higher levels. The use of such layers, or **abstractions**, is a principal technique for designing very sophisticated computer systems.

One of the most important abstractions is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special

abstraction A model that renders lower-level details of computer systems temporarily invisible in order to facilitate design of sophisticated systems.



DIMM (dual inline memory module) A small board that contains DRAM chips on both sides. SIMMs have DRAMs on only one side. Both DIMMs and SIMMs are meant to be plugged into memory slots, usually on a motherboard.

instruction set architecture

Also called architecture. An abstract interface between the hardware and the lowest level software of a machine that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

application binary interface (ABI)

The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

implementation Hardware that obeys the architecture abstraction.

FIGURE 1.10 Close-up of PC motherboard. This board uses the Intel Pentium 4 processor, which is located on the left-upper quadrant of the board. It is covered by a set of metal fins, which look like a radiator. This structure is the *heat sink*, used to help cool the chip. The main memory is contained on one or more small boards that are perpendicular to the motherboard near the middle. The DRAM chips are mounted on these boards (called **DIMMs**, for dual inline memory modules) and then plugged into the connectors. Much of the rest of the board comprises connectors for external I/O devices: audio/MIDI and parallel/serial at the right edge, two PCI card slots near the bottom, and an ATA connector used for attaching hard disks.

name: the **instruction set architecture**, or simply **architecture**, of a machine. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. Typically the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions, so that application programmers do not need to worry about such details. The combination of the basic instruction set and the operating system interface provided for application programmers is called the **application binary interface** (ABI).

An instruction set architecture allows computer designers to talk about functions independently from the hardware that performs them. For example, we can talk about the functions of a digital clock (keeping time, displaying the time, setting the alarm) independently from the clock hardware (quartz crystal, LED displays, plastic buttons). Computer designers distinguish architecture from an **implementation** of an architecture along the same lines: an implementation is hardware that obeys the architecture abstraction. These ideas bring us to another Big Picture.

A Safe Place for Data

Thus far we have seen how to input data, compute using the data, and display data. If we were to lose power to the computer, however, everything would be lost because the **memory** inside the computer is **volatile**—that is, when it loses power, it forgets. In contrast, a cassette tape for a stereo doesn't forget the recorded music when you turn off the power because the tape is magnetic and is thus a **nonvolatile memory** technology.

To distinguish between the memory used to hold programs while they are running and this nonvolatile memory used to store programs between runs, the term **primary memory** or **main memory** is used for the former, and **secondary memory** for the latter. DRAMs have dominated main memory since 1975, but **magnetic disks** have dominated secondary memory since 1965. In embedded applications, FLASH, a nonvolatile semiconductor memory is also used.

Today the primary nonvolatile storage used on all desktop and server computers is the magnetic hard disk. As Figure 1.11 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or video tape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read/write head* is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

Diameters of hard disks vary by more than a factor of 3 today, from less than 1 inch to 3.5 inches, and have been shrunk over the years to fit into new products; workstation servers, personal computers, laptops, palmtops, and digital cameras have all inspired new disk form factors. Traditionally, the widest disks have the highest performance, the smallest disks have the lowest unit cost, and the best cost per **megabyte** is usually a disk in between. Although most hard drives appear inside computers (as in Figure 1.8), hard drives can also be attached using external interfaces such as Firewire or USB.

The use of mechanical components means that access times for magnetic disks are much slower than for DRAMs: disks typically take 5–15 milliseconds, while DRAMs take 40–80 nanoseconds—making DRAMs about 100,000 times faster. Yet disks have much lower costs than DRAM for the same storage capacity because the production costs for a given amount of disk storage are lower than for the same amount of integrated circuit. In 2004, the cost per megabyte of disk is about 100 times less expensive than DRAM.

Thus there are three primary differences between magnetic disks and main memory: disks are nonvolatile because they are magnetic; they have a slower access time because they are mechanical devices; and they are cheaper per megabyte because they have very high storage capacity at a modest cost.

memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

volatile memory Storage, such as DRAM, that only retains data only if it is receiving power.

nonvolatile memory A form of memory that retains data even in the absence of a power source and that is used to store programs between runs. Magnetic disk is nonvolatile and DRAM is not.

primary memory Also called main memory. Volatile memory used to hold programs while they are running; typically consists of DRAM in today's computers.

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of magnetic disks in today's computers.

magnetic disk (also called hard disk) A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.

megabyte Traditionally 1,048,576 (2^{20}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000 (10^6) bytes.



FIGURE 1.11 A disk showing 10 disk platters and the read/write heads.

The **BIG** Picture

Both hardware and software consist of hierarchical layers, with each lower layer hiding details from the level above. This principle of *abstraction* is the way both hardware designers and software designers cope with the complexity of computer systems. One key interface between the levels of abstraction is the *instruction set architecture*—the interface between the hardware and low-level software. This abstract interface enables many *implementations* of varying cost and performance to run identical software.

Although hard drives are not removable, there are several storage technologies in use that include the following:

- Optical disks, including both compact disks (CDs) and digital video disks (DVDs), constitute the most common form of removable storage.
- Magnetic tape provides only slow serial access and has been used to back up disks, in a role now often replaced by duplicate hard drives.
- FLASH-based removable memory cards typically attach by a USB (Universal Serial Bus) connection and are often used to transfer files.
- Floppy drives and Zip drives are a version of magnetic disk technology with removable flexible disks. **Floppy disks** were the original primary storage for personal computers, but have now largely vanished.

Optical disk technology works in a completely different way than magnetic disk technology. In a CD, data is recorded in a spiral fashion, with individual bits being recorded by burning small pits—approximately 1 micron (10^{-6} meters) in diameter—into the disk surface. The disk is read by shining a laser at the CD surface and determining by examining the reflected light whether there is a pit or flat (reflective) surface. DVDs use the same approach of bouncing a laser beam off a series of pits and flat surfaces. In addition, there are multiple layers that the laser beam can be focused on, and the size of each bit is much smaller, which together yield a significant increase in capacity.

CD and DVD writers in personal computers use a laser to make the pits in the recording layer on the CD or DVD surface. This writing process is relatively slow, taking from tens of minutes (for a full CD) to close to an hour (for a full DVD). Thus, for large quantities a different technique called pressing is used, which costs only pennies per CD or DVD.

Rewritable CDs and DVDs use a different recording surface that has a crystalline, reflective material; pits are formed that are not reflective in a manner similar to that for a write-once CD or DVD. To erase the CD or DVD, the surface is heated and cooled slowly, allowing an annealing process to restore the surface recording layer to its crystalline structure. These rewritable disks are the most expensive, with write-once being cheaper; for read-only disks—used to distribute software, music, or movies—both the disk cost and recording cost are much lower.

floppy disk A portable form of secondary memory composed of a rotating mylar platter coated with a magnetic recording material.

Communicating with Other Computers

We've explained how we can input, compute, display, and save data, but there is still one missing item found in today's computers: computer networks. Just as the processor shown in Figure 1.5 on page 16 is connected to memory and I/O devices, networks connect whole computers, allowing computer users to extend

the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new machine without an optional network interface would be ridiculed. Networked computers have several major advantages:

- *Communication*: Information is exchanged between computers at high speeds.
- *Resource sharing*: Rather than each machine having its own I/O devices, devices can be shared by computers on the network.
- *Nonlocal access*: By connecting computers over long distances, users need not be near the computer they are using.

Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is the *Ethernet*. Its length is limited to about a kilometer, and the most popular version in 2004 takes about a tenth of a second to send 1 million bytes of data. Its length and speed make Ethernet useful to connect computers on the same floor of a building; hence, it is an example of what is generically called a **local area network**. Local area networks are interconnected with switches that can also provide routing services and security. **Wide area networks** cross continents and are the backbone of the Internet, which supports the World Wide Web. They are typically based on optical fibers and are leased from telecommunication companies.

local area network (LAN) A network designed to carry data within a geographically confined area, typically within a single building.

wide area network A network extended over hundreds of kilometers which can span a continent.

Networks have changed the face of computing in the last 25 years both by becoming much more ubiquitous and by dramatic increases in performance. In the 1970s, very few individuals had access to electronic mail, the Internet and Web did not exist, and physically mailing magnetic tapes was the primary way to transfer large amounts of data between two locations. In the 1970s, local area networks were almost nonexistent, and the few existing wide area networks had limited capacity and restricted access.

As networking technology improved, it became much cheaper and had a much higher capacity. For example, the first standardized local area network technology developed about 25 years ago was a version of Ethernet that had a maximum capacity (also called bandwidth) of 10 million bits per second, typically shared by tens, if not a hundred, computers. Today, local area network technology offers a capacity of from 100 million bits per second to a gigabit per second, usually shared by at most a few computers. Furthermore, 10-gigabit technology is in development! Optical communications technology has allowed similar growth in the capacity of wide area networks from hundreds of kilobits to gigabits, and from hundreds of computers connected to a worldwide network to millions of computers connected. This combination of dramatic rise in deployment of networking

combined with the increases in capacity have made network technology central to the information revolution of the last 25 years.

Recently, another innovation in networking is reshaping the way computers communicate. Wireless technology has become widely deployed, and most laptops now incorporate this technology. The ability to make a radio in the same low-cost semiconductor technology (CMOS) used for memory and microprocessors enabled a significant improvement in price, leading to an explosion in deployment. Currently available wireless technologies, called by the IEEE standard name 802.11, allow for transmission rates from 1 to less than 100 million bits per second. Wireless technology is quite a bit different from wire-based networks, since all users in an immediate area share the airwaves.

- 1. Semiconductor DRAM and disk storage differ significantly. Describe the fundamental difference for each of the following: volatility, access time, and cost.

Check Yourself

Technologies for Building Processors and Memories

Processors and memory have improved at an incredible rate because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. Figure 1.12 shows the technologies that have been used over time, with an estimate of the relative performance per unit cost for each technology. This section explores the technology that has fueled the computer industry since 1975 and will continue to do so for the foreseeable future. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. To describe the tremendous increase in the number of transistors from hundreds to

transistor An on/off switch controlled by an electric signal.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large scale integrated circuit	2,400,000
2005	Ultra large scale integrated circuit	6,200,000,000

vacuum tube An electronic component, predecessor of the transistor, that consists of a hollow glass tube about 5 to 10 cm long from which as much air has been removed as possible and which uses an electron beam to transfer data.

FIGURE 1.12 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2005 extrapolated by the authors.

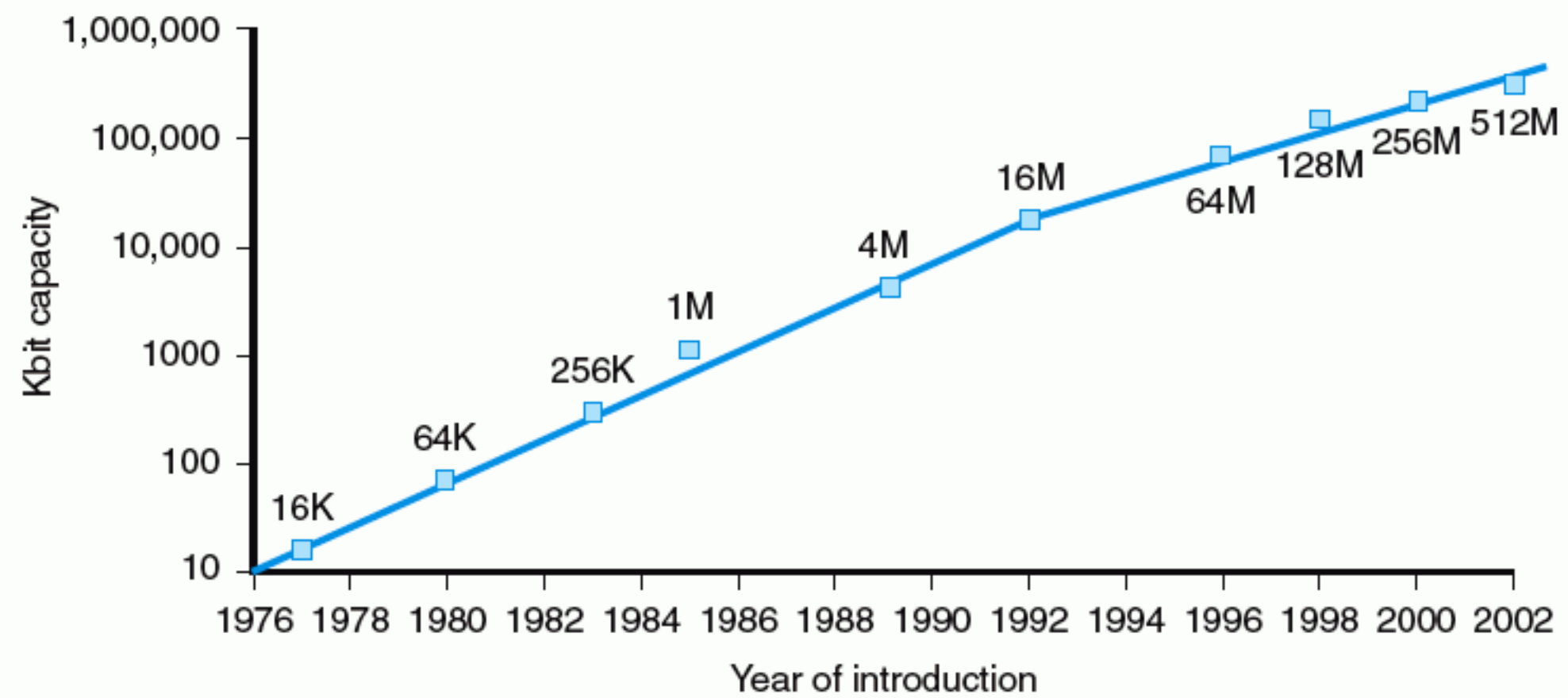


FIGURE 1.13 Growth of capacity per DRAM chip over time. The y -axis is measured in Kbits, where $K = 1024$ (2^{10}). The DRAM industry quadrupled capacity almost every 3 years, a 60% increase per year, for 20 years. This “four times every three years” estimate was called the *DRAM growth rule*. In recent years, the rate has slowed down somewhat and is somewhat closer to doubling every two years or four times every four years.

very large scale integrated (VLSI) circuit A device containing hundreds of thousands to millions of transistors.

I thought [computers] would be a universally applicable idea, like a book is. But I didn't think it would develop as fast as it did, because I didn't envision we'd be able to get as many parts on a chip as we finally got. The transistor came along unexpectedly. It all happened much faster than we expected.

J. Presper Eckert, coinventor of ENIAC, speaking in 1991

millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large scale integrated circuit**.

This rate of increasing integration has been remarkably stable. Figure 1.13 shows the growth in DRAM capacity since 1977. For 20 years, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times! This increase in transistor count for an integrated circuit is popularly known as Moore's law, which states that transistor capacity doubles every 18–24 months. Moore's law resulted from a prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel during the 1960s.

Sustaining this rate of progress for almost 40 years has required incredible innovation in the manufacturing techniques. In Section 1.4, we discuss how integrated circuits are manufactured.

1.4

Real Stuff: Manufacturing Pentium 4 Chips

Each chapter has a section entitled “Real Stuff” that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying the IBM-compatible PC, the Apple Macintosh, a common server, or an embedded computer. For this first “Real Stuff” section, we look at how integrated circuits are manufactured, with the Pentium 4 as an example.

Let's start at the beginning. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct *or* insulate under special conditions (as a switch)

Transistors fall in the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single, small package.

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.14 shows that process. The process starts with a **silicon crystal ingot**, which looks like a giant sausage. Today, ingots are 8–12 inches in diameter and about 12–24 inches long. An ingot is finely sliced into **wafers** no more than 0.1 inch thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed

silicon A natural element which is a semiconductor.

semiconductor A substance that does not conduct electricity well.

silicon crystal ingot A rod composed of a silicon crystal that is between 6 and 12 inches in diameter and about 12 to 24 inches long.

wafer A slice from a silicon ingot no more than 0.1 inch thick, used to create chips.

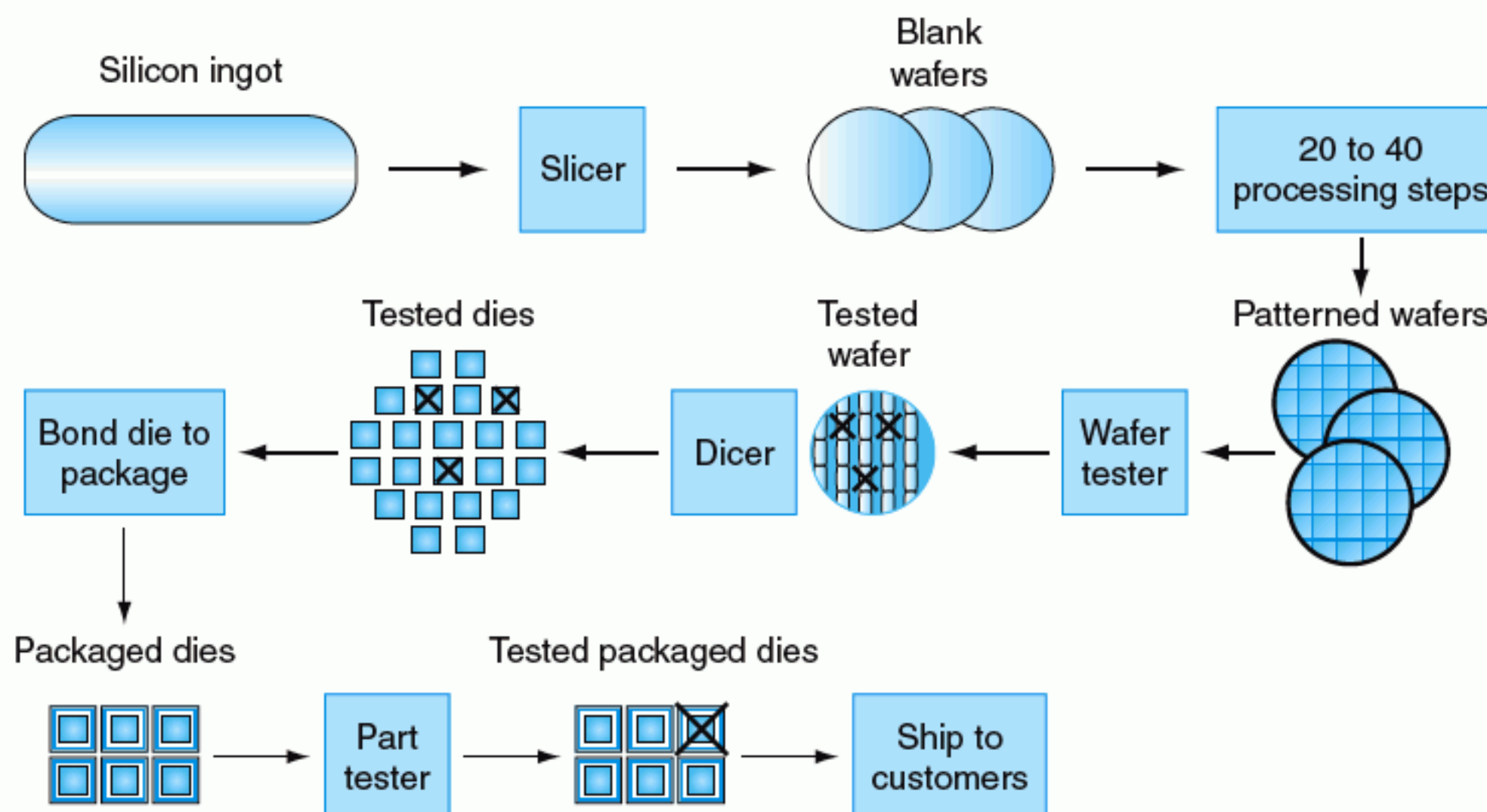


FIGURE 1.14 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.15 on page 31). These patterned wafers are then tested with a wafer tester and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9 on page 21). In this figure, one wafer produced 20 dies, of which 17 passed testing. (X means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

defect A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.


die The individual rectangular sections that are cut from a wafer, more informally known as chips.

yield The percentage of good dies from the total number of dies on the wafer.

on each wafer, creating the transistors, conductors, and insulators discussed earlier. Today's integrated circuits contain only one layer of transistors but may have from two to eight levels of metal conductor, separated by layers of insulators.

A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer. To cope with imperfection, several strategies have been used, but the simplest is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced*, into these components, called **dies** and more informally known as **chips**. Figure 1.15 is a photograph of a wafer containing Pentium 4 microprocessors before they have been diced; earlier, Figure 1.9 on page 21 showed an individual die of the Pentium 4 and its major components.

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the **yield** of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of large dies that fit on a wafer. To reduce the cost, a large die is often “shrunk” by using the next generation process, which incorporates smaller sizes for both transistors and wires. This improves the yield and the die count per wafer. (An  **Integrated Circuit Cost** section on the CD probes these issues further.)

Once you've found good dies, they are connected to the input/output pins of a package, using a process called *bonding*. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.

Another increasingly important design constraint is power. Power is a challenge for two reasons. First, power must be brought in and distributed around the chip; modern microprocessors use hundreds of pins just for power and ground! Similarly, multiple levels of interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. An Intel Pentium 4 at 3.06 GHz burns 82 watts, which must be removed from a chip whose surface area is just over 1 cm²! Figure 1.16 shows a 3.06 GHz Pentium 4 mounted on top of its heat sink, which in turn sits right next to the fan in the box shown in Figure 1.8 (on page 19)!

What determines the power consumed by an integrated circuit? Ignoring technology and circuit specifics, the power is proportional to the product of the number of transistors switched times the frequency they are switched. Thus, in general, higher clock rates or higher transistor counts lead to higher power. For example, the Intel Itanium 2 has four times the transistors of the Intel Pentium 4; although its clock rate is only one-half that of the Pentium 4, the Itanium burns 130 watts

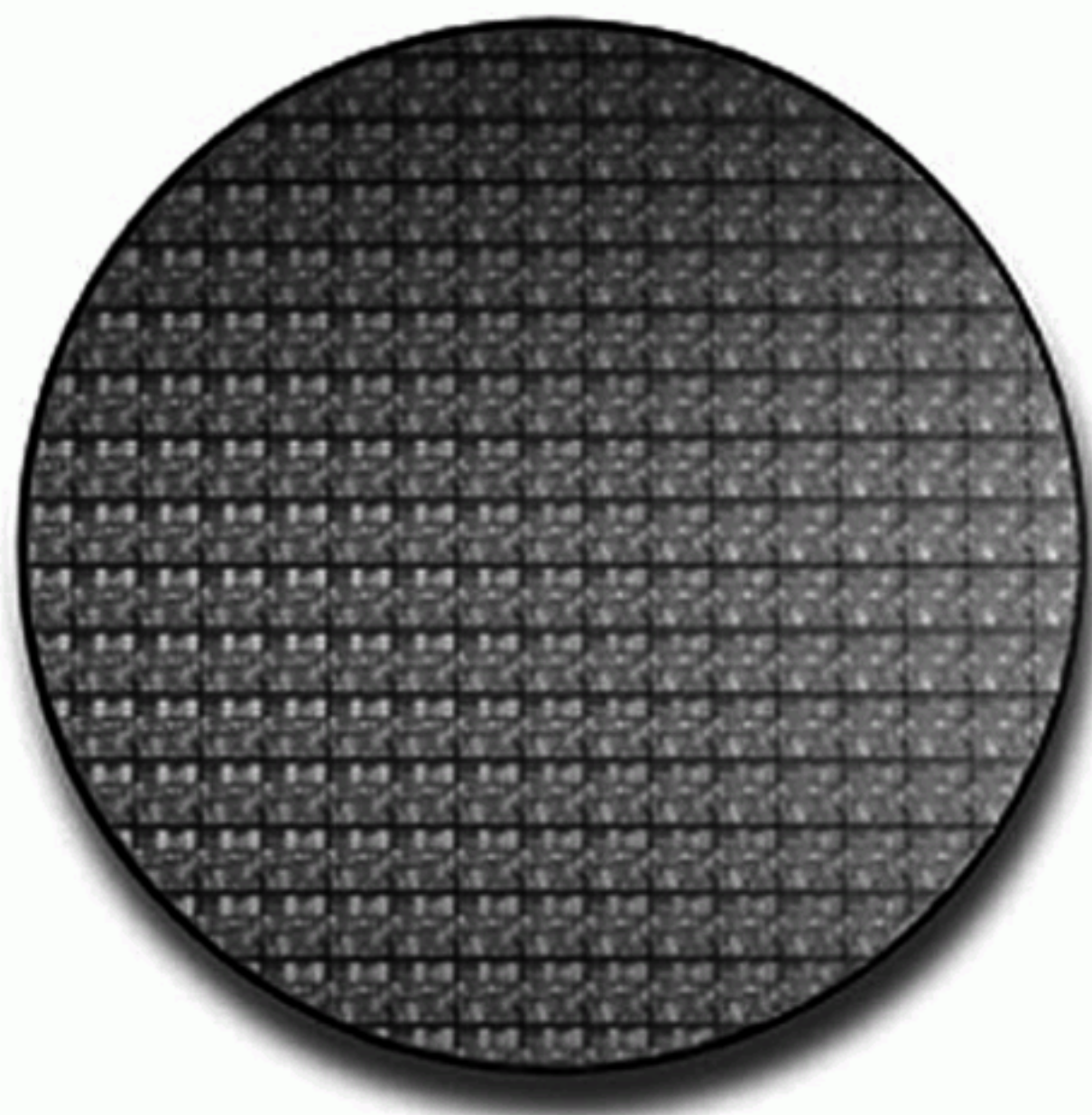


FIGURE 1.15 An 8-inch (200-mm) diameter wafer containing Intel Pentium 4 processors.

The number of Pentium dies per wafer at 100% yield is 165. Figure 1.9 on page 21 is a photomicrograph of one of these Pentium 4 dies. The die area is 250 mm^2 , and it contains about 55 million transistors. This die uses a 0.18 micron technology, which means that the smallest transistors are approximately 0.18 microns in size, although they are typically somewhat smaller than the actual feature size, which refers to the size of the transistors as “drawn” versus the final manufactured size. The Pentium 4 is also made using a more advanced 0.13 micron technology. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it’s easier to create the masks used to pattern the silicon.

compared to the 82 watts consumed by the Pentium 4. As we will see in later chapters, both performance and power consumption vary widely.

Elaboration: In CMOS (Complementary Metal Oxide Semiconductor), which is the dominant technology for integrated circuits, the primary source of power dissipation is so-called “dynamic power”—that is, power that is consumed during switching. CMOS technology, unlike earlier technologies, does not directly consume power when it is idle—hence the use of low clock rates to allow a processor to “sleep” and conserve power. The dynamic power dissipation depends on the capacitive loading of each transistor, the voltage applied, and the frequency that the transistor is switched:

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

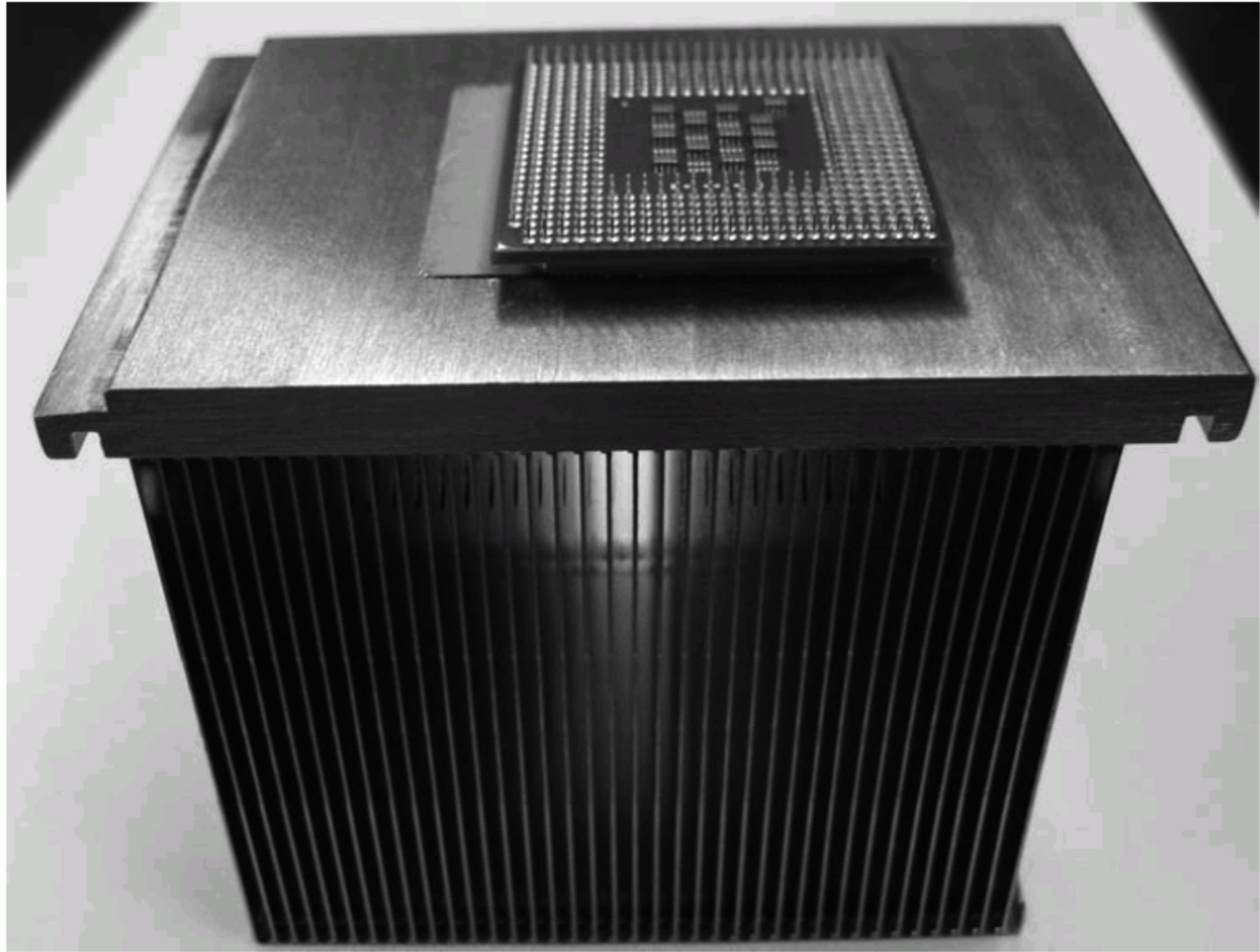


FIGURE 1.16 An Intel Pentium 4 (3.06 GHz) mounted on top of its heat sink, which is designed to remove the 82 watts generated within the die.

Power can be reduced by lowering the voltage, which typically occurs with a new generation of technology; in 20 years, voltages have gone from 5V to 1.5V, significantly reducing power. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the *fanout*) and the technology, which determines the capacitance of both wires and transistors.

Although dynamic power is the primary source of power dissipation in CMOS, static power dissipation occurs because of leakage current that flows even when a transistor is off. In 2004, leakage is probably responsible for 20–30% of the power consumption. Thus, increasing the number of transistors increases power dissipation, even if the transistors are always off. A variety of design techniques and technology innovations have been deployed to control leakage.

A key factor in determining the cost of an integrated circuit is volume. Which of the following are reasons why a chip made in high volume should cost less?

1. With high volumes, the manufacturing process can be tuned to a particular design, increasing the yield.
2. It is less work to design a high-volume part than a low-volume part.
3. The masks used to make the chip are expensive, so the cost per chip is lower for higher volumes.
4. Engineering development costs are high and largely independent of volume; thus, the development cost per die is lower with high-volume parts.
5. High-volume parts usually have smaller die sizes than low-volume parts and therefore have higher yield per wafer.

Check Yourself

1.5

Fallacies and Pitfalls

The purpose of a section on fallacies and pitfalls, which will be found in every chapter, is to explain some commonly held misconceptions that you might encounter. We call such misbeliefs *fallacies*. When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*, or easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these mistakes in the machines you may design or use.

Fallacy: Computers have been built in the same, old-fashioned way for far too long, and this antiquated model of computation is running out of steam.

For an antiquated model of computation, it surely is improving quickly. Figure 1.17 plots the top performance per year of workstations between 1987 and 2003. (Chapter 4 explains the proper way to measure performance.) The graph shows a line indicating an improvement of 1.54 per year, or doubling performance approximately every 18 months. In contrast to the statement above, computers are improving in performance faster today than at any time in their history, with over a thousandfold improvement between 1987 and 2003!

Pitfall: Ignoring the inexorable progress of hardware when planning a new machine.

Suppose you plan to introduce a machine in three years, and you claim the machine will be a terrific seller because it's three times as fast as anything available today. Unfortunately, the machine will probably sell poorly because the average

Science must begin with myths, and the criticism of myths.

Sir Karl Popper, *The Philosophy of Science*, 1957

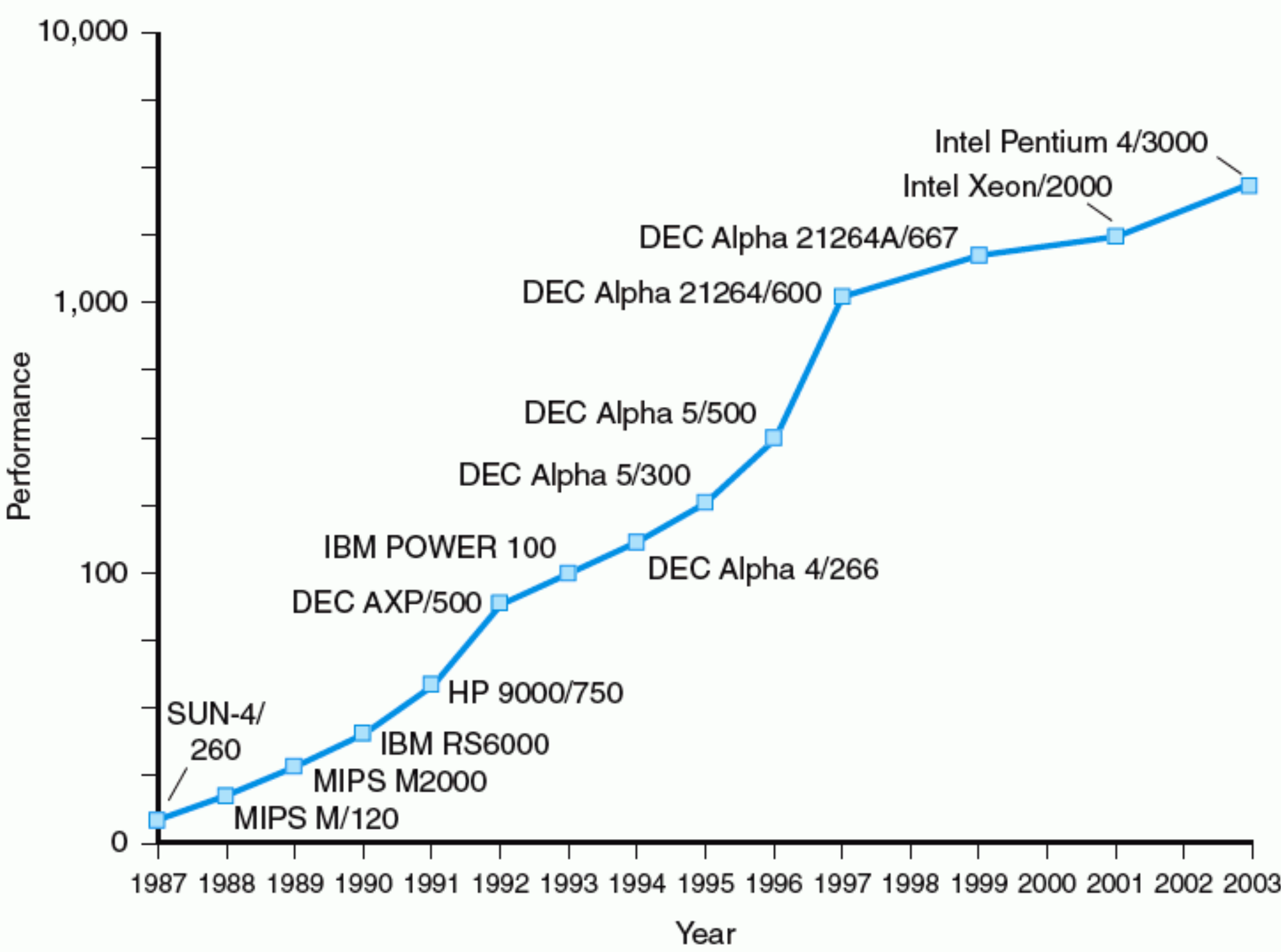


FIGURE 1.17 Performance increase of workstations, 1987–2003. Here performance is given as approximately the number of times faster than the VAX-11/780, which was a commonly used yardstick. The rate of performance improvement is between 1.5 and 1.6 times per year. These performance numbers are based on SPECint performance (see Chapter 2) and scaled over time to deal with changing benchmark sets. For processors listed with x/y after their name, x is the model number and y is the speed in megahertz.

performance growth rate for the industry will yield machines with the same performance. For example, assuming a 50% yearly growth rate in performance, a machine with performance x today can be expected to have performance $1.5^3x = 3.4x$ in three years. Your machine would have no performance advantage! Many projects within computer companies are canceled, either because they ignore this rule or because the project is completed late and the performance of the delayed machine is below the industry average. This phenomenon may occur in any industry, but rapid improvements in cost/performance make it a major concern in the computer industry.

1.6**Concluding Remarks**

Although it is difficult to predict exactly what level of cost/performance computers will have in the future, it's a safe bet that they will be much better than they are today. To participate in these advances, computer designers and programmers must understand a wider variety of issues.

Both hardware and software designers construct computer systems in hierarchical layers, with each lower layer hiding details from the level above. This principle of abstraction is fundamental to understanding today's computer systems, but it does not mean that designers can limit themselves to knowing a single technology. Perhaps the most important example of abstraction is the interface between hardware and low-level software, called the *instruction set architecture*. Maintaining the instruction set architecture as a constant enables many implementations of that architecture—presumably varying in cost and performance—to run identical software. On the downside, the architecture may preclude introducing innovations that require the interface to change.

Key technologies for modern processors are compilers and silicon. Clearly, to participate you must understand some of the characteristics of both. Equal in importance to an understanding of integrated circuit technology is an understanding of the expected rates of technological change. While silicon fuels the rapid advance of hardware, new ideas in the organization of computers have improved price/performance. Two of the key ideas are exploiting parallelism in the processor, typically via pipelining, and exploiting locality of accesses to a memory hierarchy, typically via caches.

Road Map for This Book

At the bottom of these abstractions are the five classic components of a computer: datapath, control, memory, input, and output (refer back to Figure 1.5). These five components also serve as the framework for the rest of the chapters in this book:

- *Datapath*: Chapters 3, 5, and 6
- *Control*: Chapters 5 and 6
- *Memory*: Chapter 7
- *Input*: Chapter 8
- *Output*: Chapter 8

Where . . . the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have 1,000 vacuum tubes and perhaps weigh just 1½ tons.

Popular Mechanics, March 1949

An active field of science is like an immense anthill; the individual almost vanishes into the mass of minds tumbling over each other, carrying information from place to place, passing it around at the speed of light.


Lewis Thomas, “Natural Science,” in *The Lives of a Cell*, 1974

Chapter 6 describes how processor pipelining exploits parallelism, and Chapter 7 describes how the memory hierarchy exploits locality. The remaining chapters provide the introduction and the conclusion to this material. Chapter 2 describes instruction sets—the interface between compilers and the machine—and emphasizes the role of compilers and programming languages in using the features of the instruction set. Chapter 3 describes how computers perform arithmetic operations and handle arithmetic data. Chapter 4 covers performance and thus describes how to evaluate the whole computer. Chapter 9 describes multiprocessors and is included on the CD. Appendix B, also on the CD, discusses logic design.



Historical Perspective and Further Reading


For each chapter in the text, a section devoted to a historical perspective can be found on the CD that accompanies this book. We may trace the development of an idea through a series of machines or describe some important projects, and we provide references in case you are interested in probing further.

The historical perspective for this chapter provides a background for some of the key ideas presented in this opening chapter. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By understanding the past, you may be better able to understand the forces that will shape computing in the future. Each historical perspectives section on the CD ends with suggestions for further reading, which are also collected separately on the CD under the section “[Further Reading](#).” The rest of this  [Section 1.7](#) is on the CD.

1.8

Exercises

The relative time ratings of exercises are shown in square brackets after each exercise number. On average, an exercise rated [10] will take you twice as long as one rated [5]. Sections of the text that should be read before attempting an exercise will be given in angled brackets; for example, <§1.2> means you should have read Section 1.3, “Under the Covers,” to help you solve this exercise. If the solution to an exercise depends on others, they will be listed in curly brackets; for example, {Ex.1.1} means that you should answer Exercise 1.1 before trying this exercise.

 [In More Depth](#) Exercises introduce a new topic or explore a topic in more detail. Such exercises include sufficient background to understand the concepts, as

well as exercises to explore its implication or use. The In More Depth sections appear on the CD associated with the specific chapter.

Starting in Chapter 2, you will also find For More Practice Exercises. For More Practice Exercises include additional problems intended to give the interested reader more practice in dealing with a subject. These exercises have been collected primarily from earlier editions of this book as well as exercises developed by other instructors. The For More Practice sections appear on the CD associated with the specific chapter.

Exercises 1.1 through 1.28 Find the word or phrase from the list below that best matches the description in the following questions. Use the numbers to the left of words in the answer. Each answer should be used only once.

- | | | | |
|----|-------------------------------------|----|--|
| 1 | abstraction | 15 | embedded system |
| 2 | assembler | 16 | instruction |
| 3 | bit | 17 | instruction set architecture |
| 4 | cache | 18 | local area network (LAN) |
| 5 | central processor unit (CPU) | 19 | memory |
| 6 | chip | 20 | operating system |
| 7 | compiler | 21 | semiconductor |
| 8 | computer family | 22 | server |
| 9 | control | 23 | supercomputer |
| 10 | datapath | 24 | transistor |
| 11 | desktop or personal computer | 25 | VLSI (very large scale integrated circuit) |
| 12 | Digital Video Disk (DVD) | 26 | wafer |
| 13 | defect | 27 | wide area network (WAN) |
| 14 | DRAM (dynamic random access memory) | 28 | yield |

1.1 [2] Active part of the computer, following the instructions of the programs to the letter. It adds numbers, tests numbers, controls other components, and so on.

1.2 [2] Approach to the design of hardware or software. The system consists of hierarchical layers, with each lower layer hiding details from the level above.

1.3 [2] Binary digit.

1.4 [2] Collection of implementations of the same instruction set architecture. They are usually made by the same company and vary in price and performance.

- 1.5** [2] Component of the computer where all running programs and associated data reside.
- 1.6** [2] Component of the processor that performs arithmetic operations.
- 1.7** [2] Component of the processor that tells the datapath, memory, and I/O devices what to do according to the instructions of the program.
- 1.8** [2] Computer designed for use by an individual, usually incorporating a graphics display, keyboard, and mouse.
- 1.9** [2] Computer inside another device used for running one predetermined application or collection of software.
- 1.10** [2] Computer used for running larger programs for multiple users often simultaneously and typically accessed only by a network.
- 1.11** [2] Computer network that connects a group of computers by a common transmission cable or wireless link within a small geographic area (for example, within the same floor of a building).
- 1.12** [2] Computer networks that connect computers spanning great distances, the backbone of the Internet.
- 1.13** [2] High-performance machine, costing more than \$1 million.
- 1.14** [2] Integrated circuit commonly used to construct main memory.
- 1.15** [2] Microscopic flaw in a wafer.
- 1.16** [2] Nickname for a die or integrated circuit.
- 1.17** [2] On/off switch controlled by electricity.
- 1.18** [2] Optical storage medium with a storage capacity of more than 4.7 GB. It was initially marketed for entertainment and later for computer users.
- 1.19** [2] Percentage of good dies from the total number of dies on the wafer.
- 1.20** [2] Program that converts a symbolic version of an instruction into the binary version.
- 1.21** [2] Program that manages the resources of a computer for the benefit of the programs that run on that machine.
- 1.22** [2] Program that translates from a higher-level notation to assembly language.
- 1.23** [2] Technology in which single chip that contains hundreds of thousands to millions of transistors.

1.24 [2] Single software command to a processor.

1.25 [2] Small, fast memory that acts as a buffer for the main memory.

1.26 [2] Specific interface that the hardware provides the low-level software.

1.27 [2] Substance that does not conduct electricity well but is the foundation of integrated circuits.

1.28 [2] Thin disk sliced from a silicon crystal ingot, which will be later divided into dies.

Exercises 1.29 through 1.45 Using the categories in the list below, classify the following examples. Use the letters to the left of the words in the answer. Unlike the previous exercises, answers in this group may be used more than once.

- | | | | |
|---|---------------------------------|---|-------------------|
| a | applications software | f | personal computer |
| b | high-level programming language | g | semiconductor |
| c | input device | h | supercomputer |
| d | integrated circuit | i | systems software |
| e | output device | | |

1.29 [1] Assembler

1.30 [1] C++

1.31 [1] Liquid crystal display (LCD)

1.32 [1] Compiler

1.33 [1] Cray-1

1.34 [1] DRAM

1.35 [1] IBM PC

1.36 [1] Java

1.37 [1] Scanner

1.38 [1] Macintosh

1.39 [1] Microprocessor

1.40 [1] Microsoft Word

1.41 [1] Mouse

1.42 [1] Operating system

1.43 [1] Printer

1.44 [1] Silicon

1.45 [1] Spreadsheet

1.46 [15] <§1.3> In a magnetic disk, the disks containing the data are constantly rotating. On average it should take half a revolution for the desired data on the disk to spin under the read/write head. Assuming that the disk is rotating at 7200 revolutions per minute (RPM), what is the average time for the data to rotate under the disk head? What is the average time if the disk is spinning at 10,000 revolutions per minute?

1.47 [5] <§1.3> A DVD drive, however, works in the Constant Linear Velocity (CLV) mode. The read head must interact with the concentric circles at a constant rate, whether it is accessing data from the inner or outermost portions of the disc. This is affected by varying the rotation speed of the disc, from 1600 RPM at the center, to 570 RPM at the outside. Assuming that the DVD drive reads 1.35 MB of user data per second, how many bytes can the center circle store? How many bytes can the outside circle store?

1.48 [5] <§1.3> If a computer issues 30 network requests per second and each request is on average 64 KB, will a 100 Mbit Ethernet link be sufficient?

1.49 [5] <§1.3> What kinds of networks do you use on a regular basis? What kinds of media do they use? How much bandwidth do they provide?

1.50 [15] <§1.3> End-to-end delay is an important performance metric for networks. It is the time between the point when the source starts to send data and the point when the data is completely delivered to the destination. Consider two hosts A and B, connected by a single link of rate R bps. Suppose the two hosts are separated by m meters, and suppose the propagation speed along the link is s m/sec. Host A is sending a file of size L bits to host B.

- Obtain an expression for the end-to-end delay in terms of R , L , m , and s .
- Suppose there is a router between A and B, and the data from A must be forwarded to B by the router. If the forwarding process takes t sec, then what is the end-to-end delay?
- Suppose the router is configured to provide QoS (Quality of Service) control for different kinds of data. If the data is a multimedia stream, such as video conference data, it will forward it at a shorter delay of $t/2$ sec. For other kinds of data, the delay is t sec. If host A is sending a multimedia stream of size $2L$, what is the end-to-end delay?

1.51 [15] <§§1.4, 1.5> Assume you are in a company that will market a certain IC chip. The fixed costs, including R&D, fabrication and equipments, and so on, add up to \$500,000. The cost per wafer is \$6000, and each wafer can be diced into 1500 dies. The die yield is 50%. Finally, the dies are packaged and tested, with a cost of \$10 per chip. The test yield is 90%; only those that pass the test will be sold to customers. If the retail price is 40% more than the cost, at least how many chips have to be sold to break even?

1.52 [8] <§1.6> In this exercise, you will evaluate the performance difference between two CPU architectures, CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing). Generally speaking, CISC CPUs have more complex instructions than RISC CPUs and therefore need fewer instructions to perform the same tasks. However, typically one CISC instruction, since it is more complex, takes more time to complete than a RISC instruction. Assume that a certain task needs P CISC instructions and $2P$ RISC instructions, and that one CISC instruction takes $8T$ ns to complete, and one RISC instruction takes $2T$ ns. Under this assumption, which one has the better performance?

1.53 [15] <§§1.3, 1.6> Suppose there are five computers connected together to form a local area network. The maximum data transport rate (bandwidth) that the network cable can provide is 10 Mbps. If we use a low-end device (Hub) to connect them, all the computers in the network share the 10 Mbps bandwidth. If we use a high-end device (Switch), then any two of the computers can communicate with each other without disturbing the other computers. If you want to download a 10 MB file from a remote server, which is located outside your local network, how long will it take if using a Hub? How long will it take if using a Switch? Assume the other four computers only communicate with each other, and each has a constant data rate of 2 Mbps.

1.54 [8] <§1.6> Sometimes software optimization can dramatically improve the performance of a computer system. Assume that a CPU can perform a multiplication operation in 10 ns, and a subtraction operation in 1 ns. How long will it take for the CPU to calculate the result of $d = a \times b - a \times c$? Could you optimize the equation so that it will take less time?

1.55 [8] <§§1.1–1.5> This book covers abstractions for computer systems at many different levels of detail. Pick another system with which you are familiar and write one or two paragraphs describing some of the many different levels of abstraction inherent in that system. Some possibilities include automobiles, homes, airplanes, geometry, the economy, a city, and the government. Be sure to identify both high-level and low-level abstractions.

1.56 [15] <§§1.1–1.5> A less technically inclined friend has asked you to explain how computers work. Write a detailed, one-page description for your friend.

1.57 [10] <§§1.1–1.5> In what ways do you lack a clear understanding of how computers work? Are there levels of abstraction with which you are particularly unfamiliar? Are there levels of abstraction with which you are familiar but still have specific questions about? Write at least one paragraph addressing each of these questions.

1.58 [15] <§1.3> In this exercise, you will learn more about interfaces or abstractions. For example, we can provide an abstraction for a disk like this:

Performance characteristics:

- Capacity (how much data can it store?)
- Bandwidth (how fast can data be transferred between the computer and disk?)
- Latency (how long does it take to find a specific position for access?)

Functions the interface provides:

- Read/write data
- Seek to a specific position
- Status report (is the disk ready to read/write, etc.?)

Following this pattern, please provide an abstraction for a network card.

1.59 [5] <§§1.4, 1.5>  **In More Depth:** Integrated Circuit Cost

1.60 [15] <§§1.4, 1.5>  **In More Depth:** Integrated Circuit Cost

1.61 [10] <§§1.4, 1.5>  **In More Depth:** Integrated Circuit Cost

1.62 [5] <§§1.4, 1.5>  **In More Depth:** Integrated Circuit Cost

1.63 [10] <§§1.4, 1.5>  **In More Depth:** Integrated Circuit Cost

1.64 [10] <§§1.4, 1.5>  **In More Depth:** Integrated Circuit Cost

Answers to Check Yourself

§1.1, page 10: Discussion questions: lots of answers are acceptable.

§1.3, page 27: Disk memory: nonvolatile, long access time (milliseconds), and cost \$2–4/GB. Semiconductor memory: volatile, short access time (nanoseconds), and cost \$200–400/GB.

§1.4, page 33: 1, 3, and 4 are valid reasons.

Computers in the Real World

Information Technology for the 4 Billion without IT

Throughout this book you will see sections entitled “Computers in the Real World.” These sections describe compelling uses for computers outside of their typical functions in office automation and data processing. The goal of these sections is to illustrate the diversity of uses for information technology.

Problem to solve: Make information technology available to the rest of humanity, such as farmers in rural villages, beyond a multilingual character set like Unicode.

Solution: Develop a computer, software, and a communication system for a rural farming village. However, there is no electricity, no telephone, no technical support, and the villagers do not read English.

The Jhai Foundation took on this challenge for five villages in the Hin Heup district of Laos. This American-Lao foundation was founded to raise the standard of living for rural Laos by developing an export economy. It also built schools, installed wells, and started a weaving cooperative. When asked

what they wanted next, villagers said they wanted access to the Internet! First, they wanted to learn the prices before taking their crops to the nearest market, which is 35 kilometers away. They could also learn about the market abroad to make better decisions on what crops to grow and to increase their bargaining power when it was time to sell them. Second, they wanted to use Internet telephony to talk to relatives in Laos and beyond.

The goal was “a rugged computer and printer assembled from off-the-shelf components that draws less than 20 watts in normal use—less than 70 watts when the printer is printing—and that can survive dirt, heat, and immersion in water.”

The resulting Jhai PC design uses flash memory instead of a disk drive, thereby eliminating moving parts from the PC to make it more rugged and easier to maintain. Rather than use a power-hungry cathode ray tube, it has a liquid-crystal display. To lower costs and power, it uses an 80486 microprocessor. The power is supplied by a car battery, which can be charged by a turning bicycle crank. An old




A Laotian villager who wanted access to the Internet.

dot matrix printer completes the hardware, bringing the cost to about \$400. The operating system is Linux, and the applications are accounting, email, and letter writing, which expatriates are tailoring to the Lao language.

The communication solution is to adapt the WiFi (IEEE 802.11b) wireless network (see Chapter 8). The plan is to boost the signal using larger antennas and then place repeater stations on the hilltops between the village and the market city. These repeaters get their power from solar cells. The local phone system ties to it at the far end, which completes the connection to the Internet. Twenty-five volunteers in Silicon Valley are developing this Jhai PC network.

An alternative attempt is the *simputer*, which stands for “simple, inexpensive, multi-lingual computer.” Indian computer scientists

designed this personal digital assistant, which is similar to the Palm Pilot, to meet the needs of villagers in third world countries. Input is through a touch screen and speech recognition so that people need not be able to write to use it. It uses three AAA batteries, which last 3 to 4 hours. The cost is \$250, and there is no special solution for communication. It’s unclear whether villagers in the developing world would spend \$250 on a PDA, where even batteries are a luxury.

To learn more see these references on the  library

“Making the Web world-wide,” *The Economist*, September 26, 2002, www.jhai.org/economist

The Jhai Foundation, www.jhai.org/

“Computers for the Third World,” *Scientific American*, October 2002






Indian villager using the Simputer.

2

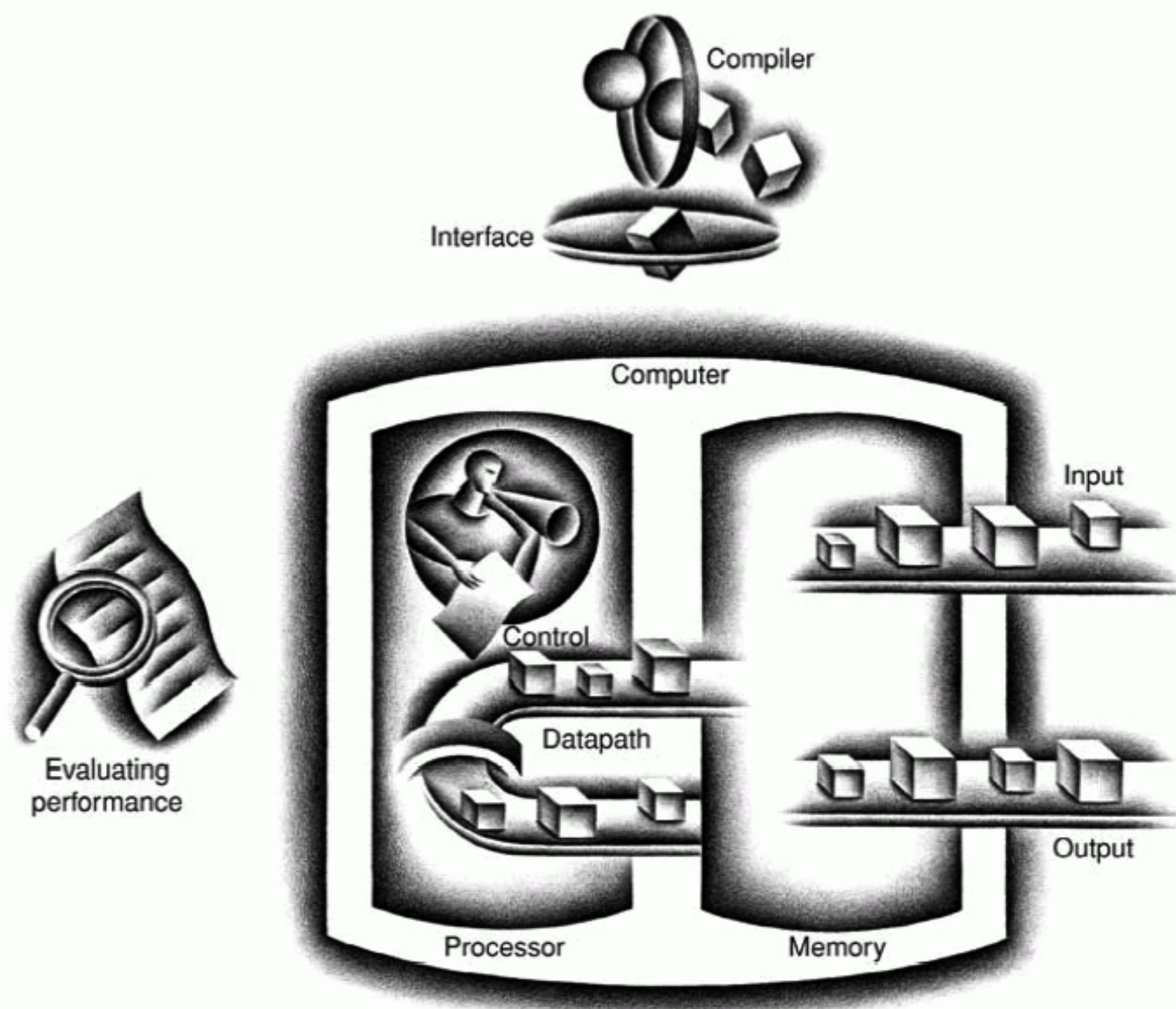
Instructions: Language of the Computer

*I speak Spanish to God,
Italian to women,
French to men,
and German to my horse.*

Charles V, King of France
1337–1380

2.1	Introduction	48
2.2	Operations of the Computer Hardware	49
2.3	Operands of the Computer Hardware	52
2.4	Representing Instructions in the Computer	60
2.5	Logical Operations	68
2.6	Instructions for Making Decisions	72
2.7	Supporting Procedures in Computer Hardware	79
2.8	Communicating with People	90
2.9	MIPS Addressing for 32-Bit Immediates and Addresses	95
2.10	Translating and Starting a Program	106
2.11	How Compilers Optimize	116
 2.12	How Compilers Work: An Introduction	121
2.13	A C Sort Example to Put It All Together	121
 2.14	Implementing an Object-Oriented Language	130
2.15	Arrays versus Pointers	130
2.16	Real Stuff: IA-32 Instructions	134
2.17	Fallacies and Pitfalls	143
2.18	Concluding Remarks	145
 2.19	Historical Perspective and Further Reading	147
2.20	Exercises	147

The Five Classic Components of a Computer



2.1

Introduction

instruction set The vocabulary of commands understood by a given architecture.

To command a computer's hardware, you must speak its language. The words of a computer's language are called *instructions*, and its vocabulary is called an **instruction set**. In this chapter, you will see the instruction set of a real computer, both in the form written by humans and in the form read by the computer. We introduce instructions in a top-down fashion. Starting from a notation that looks like a restricted programming language, we refine it step-by-step until you see the real language of a real computer. Chapter 3 continues our downward descent, unveiling the representation of integer and floating-point numbers and the hardware that operates on them.

You might think that the languages of computers would be as diverse as those of humans, but in reality computer languages are quite similar, more like regional dialects than like independent languages. Hence, once you learn one, it is easy to pick up others. This similarity occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all computers must provide. Moreover, computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost. This goal is time-honored; the following quote was written before you could buy a computer, and it is as true today as it was in 1947:

It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations. . . . The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

Burks, Goldstine, and von Neumann, 1947

The “simplicity of the equipment” is as valuable a consideration for computers of the 2000s as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in hardware and the relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language; Section 2.14 shows how these would change for an object-oriented language like Java.

By learning how to represent instructions, you will also discover the secret of computing: the **stored-program concept**. Moreover you will exercise your “foreign language” skills by writing programs in the language of the computer and running them on the simulator that comes with this book. You will also see the impact of programming languages and compiler optimization on performance. We conclude with a look at the historical evolution of instruction sets and an overview of other computer dialects.

The chosen instruction set comes from MIPS, which is typical of instruction sets designed since the 1980s. Almost 100 million of these popular microprocessors were manufactured in 2002, and they are found in products from ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Texas Instruments, and Toshiba, among others.

We reveal the MIPS instruction set a piece at a time, giving the rationale along with the computer structures. This top-down, step-by-step tutorial weaves the components with their explanations, making assembly language more palatable. To keep the overall picture in mind, each section ends with a figure summarizing the MIPS instruction set revealed thus far, highlighting the portions presented in that section.

stored-program concept The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

2.2

Operations of the Computer Hardware

Every computer must be able to perform arithmetic. The MIPS assembly language notation

```
add a, b, c
```

instructs a computer to add the two variables *b* and *c* and to put their sum in *a*.

This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of variables *b*, *c*, *d*, and *e* into variable *a*. (In this section we are being deliberately vague about what a “variable” is; in the next section we’ll explain in detail.)

The following sequence of instructions adds the four variables:

```
add a, b, c      # The sum of b and c is placed in a.
add a, a, d      # The sum of b, c, and d is now in a.
add a, a, e      # The sum of b, c, d, and e is now in a.
```

There must certainly be instructions for performing the fundamental arithmetic operations.

Burks, Goldstine, and von Neumann, 1947

Thus, it takes three instructions to take the sum of four variables.

The words to the right of the sharp symbol (*#*) on each line above are *comments* for the human reader, and the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.

The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum. Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number. This situation illustrates the first of four underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

We can now show, in the two examples that follow, the relationship of programs written in higher-level programming languages to programs in this more primitive notation.

EXAMPLE

Compiling Two C Assignment Statements into MIPS

This segment of a C program contains the five variables *a*, *b*, *c*, *d*, and *e*. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c ;  
d = a - e ;
```

The translation from C to MIPS assembly language instructions is performed by the *compiler*. Show the MIPS code produced by a compiler.

A MIPS instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two MIPS assembly language instructions:

```
add a, b, c  
sub d, a, e
```

ANSWER

Compiling a Complex C Assignment into MIPS

A somewhat complex statement contains the five variables f, g, h, i, and j:

```
f = (g + h) - (i + j);
```

What might a C compiler produce?

The compiler must break this statement into several assembly instructions since only one operation is performed per MIPS instruction. The first MIPS instruction calculates the sum of g and h. We must place the result somewhere, so the compiler creates a temporary variable, called t0:

```
add t0,g,h # temporary variable t0 contains g + h
```

Although the next operation is subtract, we need to calculate the sum of i and j before we can subtract. Thus, the second instruction places the sum i and j in another temporary variable created by the compiler, called t1:

```
add t1,i,j # temporary variable t1 contains i + j
```

Finally, the subtract instruction subtracts the second sum from the first and places the difference in the variable f, completing the compiled code:

```
sub f,t0,t1 # f gets t0 - t1, which is (g + h)-(i + j)
```

EXAMPLE

ANSWER

Figure 2.1 summarizes the portions of MIPS assembly language described in this section. These instructions are symbolic representations of what the MIPS processor actually understands. In the next few sections, we will evolve this symbolic representation into the real language of MIPS, with each step making the symbolic representation more concrete.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add a,b,c	a = b + c	Always three operands
	subtract	sub a,b,c	a = b - c	Always three operands

FIGURE 2.1 MIPS architecture revealed in Section 2.2. The real computer operands will be unveiled in the next section. Highlighted portions in such summaries show MIPS assembly language structures introduced in this section; for this first figure, all is new.

Check Yourself

For a given function, which programming language likely takes the most lines of code? Put the three representations below in order.

1. Java
2. C
3. MIPS assembly language

Elaboration: To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called *Java bytecodes*, which is quite different from the MIPS instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like MIPS. Because this compilation is normally done much later than for C programs, such Java compilers are often called *Just-In-Time* (JIT) compilers. Section 2.10 shows how JITs are used later than C compilers in the startup process, and Section 2.13 shows the performance consequences of compiling versus interpreting Java programs. The Java examples in this chapter skip the Java bytecode step and just show the MIPS code that are produced by a compiler.

2.3

Operands of the Computer Hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called *registers*. Registers are the bricks of computer construction: registers are primitives used in hardware design that are also visible to the programmer when the computer is completed. The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name **word** in the MIPS architecture.

word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers. MIPS has 32 registers. (See Section 2.19 for the history of the number of registers.) Thus, continuing in our top-down, stepwise evolution of the symbolic representation of the MIPS language, in this section we have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-bit registers.

The reason for the limit of 32 registers may be found in the second of our four underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Guidelines such as “smaller is faster” are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer’s desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format, as Section 2.4 demonstrates.

Chapters 5 and 6 show the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is key to program performance.

Although we could simply write instructions using numbers for registers, from 0 to 31, the MIPS convention is to use two-character names following a dollar sign to represent a register. Section 2.7 will explain the reasons behind these names. For now, we will use `$s0`, `$s1`, ... for registers that correspond to variables in C and Java programs and `$t0`, `$t1`, ... for temporary registers needed to compile the program into MIPS instructions.

Compiling a C Assignment Using Registers

It is the compiler’s job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS code?

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, `$t0` and `$t1`, which correspond to the temporary variables above:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1  # f gets $t0 - $t1, which is (g + h)-(i + j)
```

EXAMPLE

ANSWER

Memory Operands

Programming languages have simple variables that contain single data elements as in these examples, but they also have more complex data structures—arrays and structures. These complex data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access such large structures?

Recall the five components of a computer introduced in Chapter 1 and depicted on page 47. The processor can keep only a small amount of data in registers, but computer memory contains millions of data elements. Hence, data structures (arrays and structures) are kept in memory.

As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure 2.2, the address of the third data element is 2, and the value of Memory[2] is 10.

The data transfer instruction that copies data from memory to a register is traditionally called *load*. The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The actual MIPS name for this instruction is `lw`, standing for *load word*.

data transfer instruction A command that moves data between memory and registers.

address A value used to delineate the location of a specific data element within a memory array.

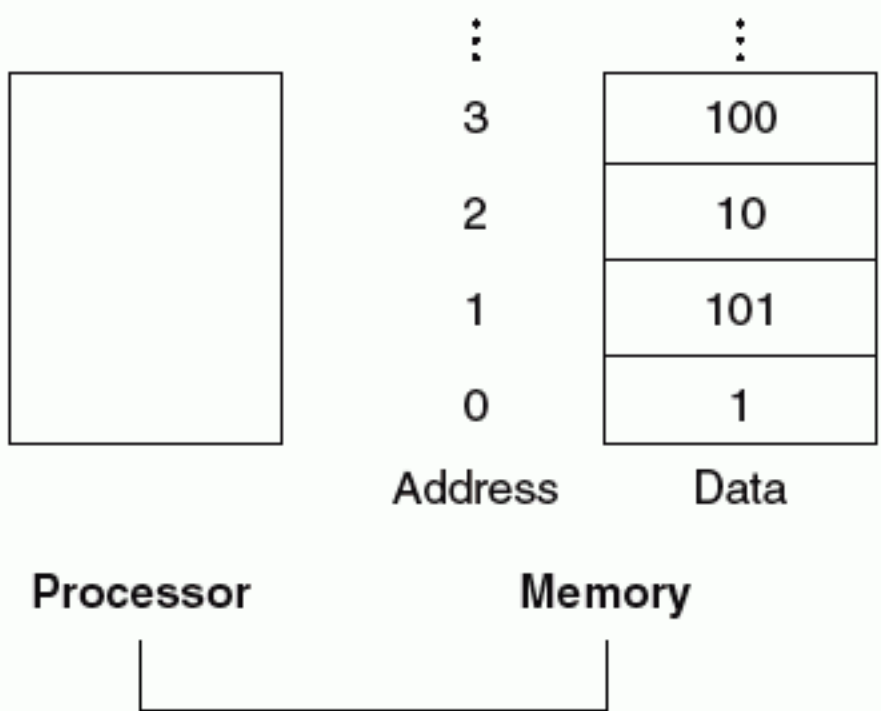


FIGURE 2.2 Memory addresses and contents of memory at those locations. This is a simplification of the MIPS addressing; Figure 2.3 shows the actual MIPS addressing for sequential word addresses in memory.

Compiling an Assignment When an Operand Is in Memory

Let's assume that *A* is an array of 100 words and that the compiler has associated the variables *g* and *h* with the registers *\$s1* and *\$s2* as before. Let's also assume that the starting address, or *base address*, of the array is in *\$s3*. Compile this C assignment statement:

```
g = h + A[8];
```

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer *A[8]* to a register. The address of this array element is the sum of the base of the array *A*, found in register *\$s3*, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. Based on Figure 2.2, the first compiled instruction is

```
lw    $t0,8($s3) # Temporary reg $t0 gets A[8]
```

(On the next page we'll make a slight adjustment to this instruction, but we'll use this simplified version for now.) The following instruction can operate on the value in *\$t0* (which equals *A[8]*) since it is in a register. The instruction must add *h* (contained in *\$s2*) to *A[8]* (*\$t0*) and put the sum in the register corresponding to *g* (associated with *\$s1*):

```
add   $s1,$s2,$t0 # g = h + A[8]
```

The constant in a data transfer instruction is called the *offset*, and the register added to form the address is called the *base register*.

EXAMPLE

ANSWER

Hardware
Software
Interface

alignment restriction

A requirement that data be aligned in memory on natural boundaries

In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

Since 8-bit *bytes* are useful in many programs, most architectures address individual bytes. Therefore, the address of a word matches the address of one of the 4 bytes within the word. Hence, addresses of sequential words differ by 4. For example, Figure 2.3 shows the actual MIPS addresses for Figure 2.2; the byte address of the third word is 8.

In MIPS, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**, and many architectures have it. (Chapter 5 suggests why alignment leads to faster data transfers.)

Computers divide into those that use the address of the leftmost or “big end” byte as the word address versus those that use the rightmost or “little end” byte. MIPS is in the *Big Endian* camp. (Appendix A, page A-43, shows the two options to number bytes in a word.)

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register \$s3 must be 4×8 , or 32*, so that the load address will select $A[8]$ and not $A[8/4]$. (See the related pitfall of page 144 of Section 2.17.)

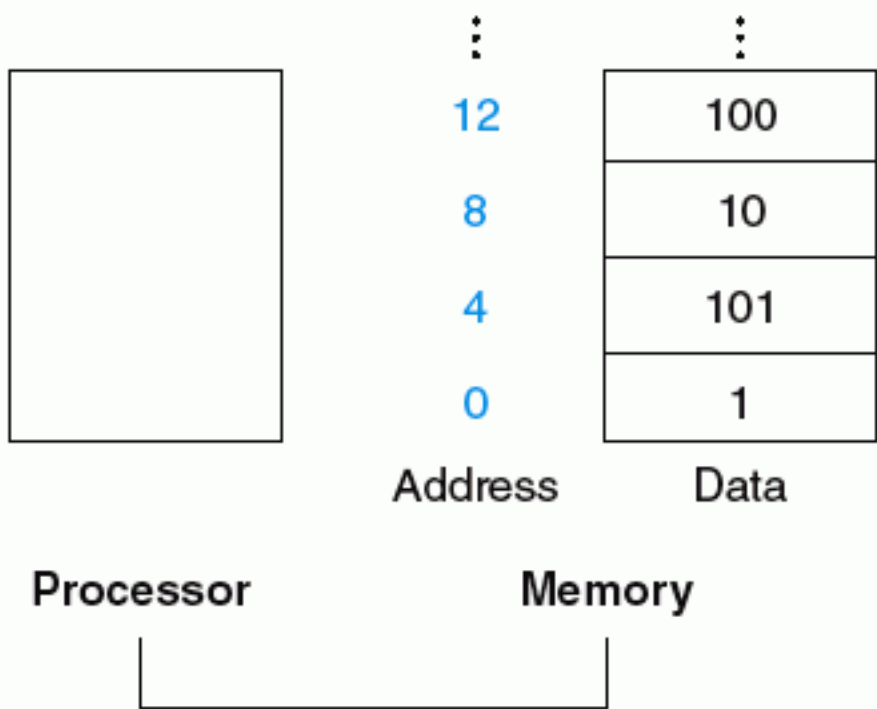


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of four: there are four bytes in a word.

The instruction complementary to load is traditionally called *store*; it copies data from a register to memory. The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then offset to select the array element, and finally the base register. Once again, the MIPS address is specified in part by a constant and in part by the contents of a register. The actual MIPS name is *sw*, standing for *store word*.

Compiling Using Load and Store

Assume variable *h* is associated with register *\$s2* and the base address of the array *A* is in *\$s3*. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select *A[8]*, and the add instruction places the sum in *\$t0*:

```
lw    $t0,32($s3)    # Temporary reg $t0 gets A[8]
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into *A[12]*, using 48 as the offset and register *\$s3* as the base register.

```
sw    $t0,48($s3)    # Stores h + A[8] back into A[12]
```

EXAMPLE

ANSWER

Constant or Immediate Operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the MIPS arithmetic instructions have a constant as an operand when running the SPEC2000 benchmarks.

Hardware Software Interface

Many programs have more variables than computers have registers. Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory. The process of putting less commonly used variables (or those needed later) into memory is called *spilling* registers.

The hardware principle relating size and speed suggests that memory must be slower than registers since registers are smaller. This is indeed the case; data accesses are faster if data is in registers instead of memory.

Moreover, data is more useful when in a register. A MIPS arithmetic instruction can read two registers, operate on them, and write the result. A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it.

Thus, MIPS registers take both less time to access *and* have higher throughput than memory—a rare combination—making data in registers both faster to access and simpler to use. To achieve highest performance, compilers must use registers efficiently.

Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register \$s3, we could use the code

```
lw    $t0, AddrConstant4($s1) # $t0 = constant 4
add   $s3,$s3,$t0             # $s3 = $s3 + $t0 ($t0 == 4)
```

assuming that AddrConstant4 is the memory address of the constant 4.

An alternative that avoids the load instruction is to offer versions of the arithmetic instructions in which one operand is a constant. This quick add instruction with one constant operand is called *add immediate* or *addi*. To add 4 to register \$s3, we just write

```
addi   $s3,$s3,4              # $s3 = $s3 + 4
```

Immediate instructions illustrate the third hardware design principle, first mentioned in the Fallacies and Pitfalls of Chapter 1:

Design Principle 3: Make the common case fast.

Constant operands occur frequently, and by including constants inside arithmetic instructions, they are much faster than if constants were loaded from memory.

MIPS operands		
Name	Example	Comments
32 registers	<code>\$s0, \$s1, . . . , \$t0, \$t1, . . .</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2 ³⁰ memory words	<code>Memory[0], Memory[4], . . . , Memory[4294967292]</code>	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	<code>add \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 + \$s3</code>	Three operands; data in registers
	subtract	<code>sub \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 - \$s3</code>	Three operands; data in registers
	add immediate	<code>addi \$s1,\$s2,100</code>	<code>\$s1 = \$s2 + 100</code>	Used to add constants
Data transfer	load word	<code>lw \$s1,100(\$s2)</code>	<code>\$s1 = Memory[\$s2 + 100]</code>	Data from memory to register
	store word	<code>sw \$s1,100(\$s2)</code>	<code>Memory[\$s2 + 100] = \$s1</code>	Data from register to memory

FIGURE 2.4 MIPS architecture revealed through Section 2.3. Highlighted portions show MIPS assembly language structures introduced in Section 2.3.

Figure 2.4 summarizes the portions of the symbolic representation of the MIPS instruction set described in this section. Load word and store word are the instructions that copy words between memory and registers in the MIPS architecture. Other brands of computers use instructions along with load and store to transfer data. An architecture with such alternatives is the Intel IA-32, described in Section 2.16.

Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

1. Very fast: They increase as fast as Moore’s law, which predicts doubling the number of transistors on a chip every 18 months.
2. Very slow: Since programs are usually distributed in the language of the computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable.

Elaboration: Although the MIPS registers in this book are 32 bits wide, there is a 64-bit version of the MIPS instruction set with 32 64-bit registers. To keep them straight, they are officially called MIPS-32 and MIPS-64. In this chapter, we use a subset of MIPS-32. Appendix D shows the differences between MIPS-32 and MIPS-64.

Check Yourself

The MIPS offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in Section 2.13.

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the *index register*. Today's memories are much larger and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

Section 2.4 explains that since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

2.4

Representing Instructions in the Computer

We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions. First, let's quickly review how a computer represents numbers.

Humans are taught to think in base 10, but numbers may be represented in any base. For example, 123 base 10 = 1111011 base 2.

Numbers are kept in computer hardware as a series of high and low electronic signals, and so they are considered base 2 numbers. (Just as base 10 numbers are called *decimal* numbers, base 2 numbers are called *binary* numbers.) A single digit of a binary number is thus the "atom" of computing, since all information is composed of **binary digits** or *bits*. This fundamental building block can be one of two values, which can be thought of as several alternatives: high or low, on or off, true or false, or 1 or 0.

Instructions are also kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.

Since registers are part of almost all instructions, there must be a convention to map register names into numbers. In MIPS assembly language, registers \$s0 to \$s7 map onto registers 16 to 23, and registers \$t0 to \$t7 map onto registers 8 to 15. Hence, \$s0 means register 16, \$s1 means register 17, \$s2 means register 18, ..., \$t0 means register 8, \$t1 means register 9, and so on. We'll describe the convention for the rest of the 32 registers in the following sections.

binary digit Also called binary bit. One of the two numbers in base 2, 0 or 1, that are the components of information.

Translating a MIPS Assembly Instruction into a Machine Instruction

Let’s do the next step in the refinement of the MIPS language as an example. We’ll show the real MIPS language version of the instruction represented symbolically as

```
add $t0,$s1,$s2
```

first as a combination of decimal numbers and then of binary numbers.

The decimal representation is

0	17	18	8	0	32
---	----	----	---	---	----

Each of these segments of an instruction is called a *field*. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation (17 = \$s1), and the third field gives the other source operand for the addition (18 = \$s2). The fourth field contains the number of the register that is to receive the sum (8 = \$t0). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register \$s1 to register \$s2 and places the sum in register \$t0.

This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

To distinguish it from assembly language, we call the numeric version of instructions **machine language** and a sequence of such instructions *machine code*.

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this MIPS instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all MIPS instructions are 32 bits long.

It would appear that you would now be reading and writing long, tedious strings of binary numbers. We avoid that tedium by using a higher base than binary that con-

EXAMPLE

ANSWER

machine language Binary representation used for communication within a computer system.

instruction format A form of representation of an instruction composed of fields of binary numbers.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0 _{hex}	0000 _{two}	4 _{hex}	0100 _{two}	8 _{hex}	1000 _{two}	c _{hex}	1100 _{two}
1 _{hex}	0001 _{two}	5 _{hex}	0101 _{two}	9 _{hex}	1001 _{two}	d _{hex}	1101 _{two}
2 _{hex}	0010 _{two}	6 _{hex}	0110 _{two}	a _{hex}	1010 _{two}	e _{hex}	1110 _{two}
3 _{hex}	0011 _{two}	7 _{hex}	0111 _{two}	b _{hex}	1011 _{two}	f _{hex}	1111 _{two}

FIGURE 2.5 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

hexadecimal Numbers in base 16.

verts easily into binary. Since almost all computer data sizes are multiples of 4, **hexadecimal** (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 2.5 converts hexadecimal to binary, and vice versa.

Because we frequently deal with different number bases, to avoid confusion we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation `0xnnnn` for hexadecimal numbers.

EXAMPLE

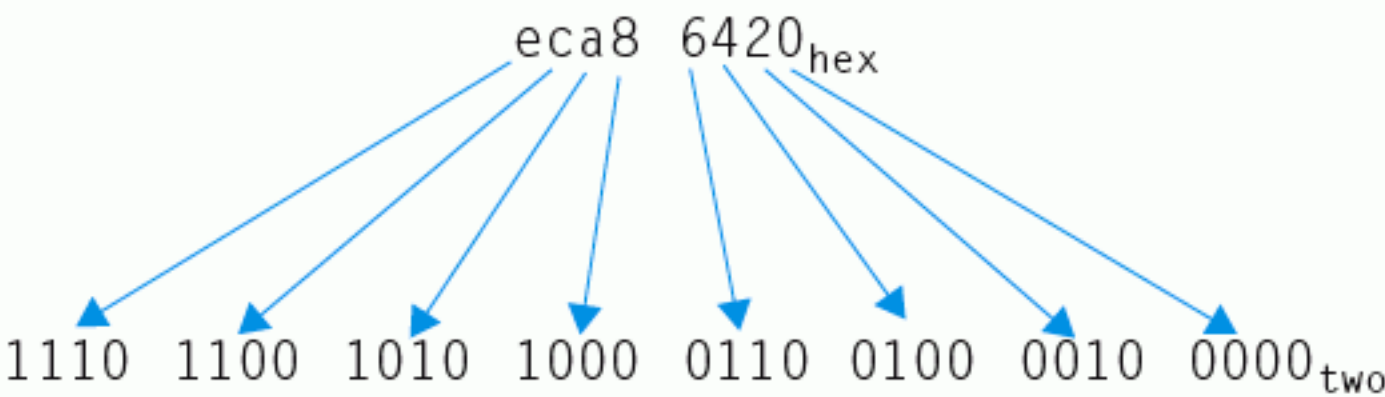
ANSWER

Binary-to-Hexadecimal and Back

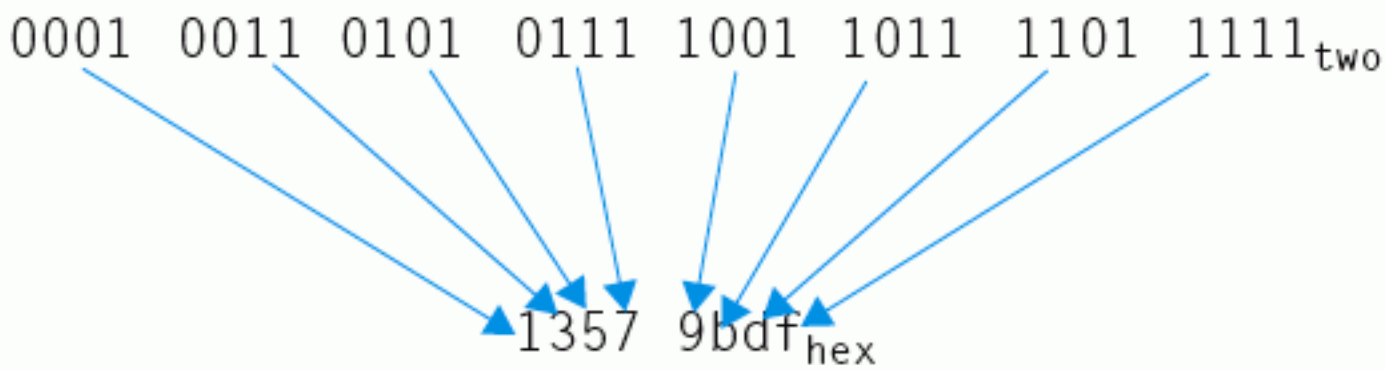
Convert the following hexadecimal and binary numbers into the other base:
eca8 6420_{hex}

0001 0011 0101 0111 1001 1011 1101 1111_{two}

Just a table lookup one way:



And then the other direction too:



MIPS Fields

MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:

- *op*: Basic operation of the instruction, traditionally called the **opcode**.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (Section 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
- *funct*: Function. This field selects the specific variant of the operation in the *op* field and is sometimes called the *function code*.

opcode The field that denotes the operation and format of an instruction.

A problem occurs when an instruction needs longer fields than those shown above. For example, the load word instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the constant within the load word instruction would be limited to only 2⁵ or 32. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 32. This 5-bit field is too small to be useful.

Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This leads us to the final hardware design principle:

Design Principle 4: Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register *rs*. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$. (Chapter 3 explains how to represent negative numbers.) We see that more than 32 registers would be difficult in this format, as the *rs* and *rt* fields would each need another bit, making it harder to fit everything in one word.

Let’s look at the load word instruction from page 57:

```
lw    $t0,32($s3)    # Temporary reg $t0 gets A[8]
```

Here, 19 (for *\$s3*) is placed in the *rs* field, 8 (for *\$t0*) is placed in the *rt* field, and 32 is placed in the address field. Note that the meaning of the *rt* field has changed for this instruction: in a load word instruction, the *rt* field specifies the *destination* register, which receives the result of the load.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first three fields of the R-type and I-type formats are the same size and have the same names; the fourth field in I-type is equal to the length of the last three fields of R-type.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (*op*) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type). Figure 2.6 shows the numbers used in each field for the MIPS instructions covered through Section 2.3.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immediate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.6 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the *op* field; the hardware uses the *funct* field to decide the variant of the operation: add (32) or subtract (34).

Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

For convenience, let's first represent the machine language instructions using decimal numbers. From Figure 2.6, we can determine the three machine language instructions:

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

The `lw` instruction is identified by 35 (see Figure 2.6) in the first field (`op`). The base register 9 (`$t1`) is specified in the second field (`rs`), and the destination register 8 (`$t0`) is specified in the third field (`rt`). The offset to select `A[300]` ($1200 = 300 \times 4$) is found in the final field (`address`).

The `add` instruction that follows is specified with 0 in the first field (`op`) and 32 in the last field (`funct`). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to `$s2`, `$t0`, and `$t0`.

EXAMPLE

ANSWER

The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

The binary equivalent to the decimal form is the following (1200 in base 10 is 0000 0100 1011 0000 base 2):

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Note the similarity of the binary representations of the first and last instructions. The only difference is in the third bit from the left.

Figure 2.7 summarizes the portions of MIPS assembly language described in this section. As we shall see in Chapters 5 and 6, the similarity of the binary representations of related instructions simplifies hardware design. These instructions are another example of regularity in the MIPS architecture.

Check Yourself

Why doesn't MIPS have a subtract immediate instruction?

- 1. Negative constants appear much less frequently in C and Java, so they are not the common case and do not merit special support.
- 2. Since the immediate field holds both negative and positive constants, add immediate with a negative number is equivalent to subtract immediate with a positive number, so subtract immediate is superfluous.

The BIG Picture

Today's computers are built on two key principles:

- 1. Instructions are represented as numbers.
- 2. Programs are stored in memory to be read or written, just like numbers.

These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 2.8 shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code. One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

MIPS operands

Name	Example	Comments
32 registers	<code>\$s0, \$s1, ..., \$s7</code> <code>\$t0, \$t1, ..., \$t7</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers <code>\$s0-\$s7</code> map to 16–23 and <code>\$t0-\$t7</code> map to 8–15.
2^{30} memory words	<code>Memory[0],</code> <code>Memory[4], ... ,</code> <code>Memory[4294967292]</code>	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	<code>add \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 + \$s3</code>	Three operands; data in registers
	subtract	<code>sub \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 - \$s3</code>	Three operands; data in registers
Data transfer	load word	<code>lw \$s1,100(\$s2)</code>	<code>\$s1 = Memory[\$s2 + 100]</code>	Data from memory to register
	store word	<code>sw \$s1,100(\$s2)</code>	<code>Memory[\$s2 + 100] = \$s1</code>	Data from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	<code>add \$s1,\$s2,\$s3</code>
sub	R	0	18	19	17	0	34	<code>sub \$s1,\$s2,\$s3</code>
addi	I	8	18	17	100			<code>addi \$s1,\$s2,100</code>
lw	I	35	18	17	100			<code>lw \$s1,100(\$s2)</code>
sw	I	43	18	17	100			<code>sw \$s1,100(\$s2)</code>
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.7 MIPS architecture revealed through Section 2.4. Highlighted portions show MIPS machine language structures introduced in Section 2.4. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; *shamt* field, which Section 2.5 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format keeps the last 16 bits as a single *address* field.

Elaboration: Representing decimal numbers in base 2 gives an easy way to represent positive integers in computer words. Chapter 3 explains how to represent negative numbers, but for now take it on faith that a 32-bit word can represent integers between -2^{31} and $+2^{31} - 1$ or $-2,147,483,648$ to $+2,147,483,647$, and the 16-bit constant field really holds -2^{15} to $+2^{15} - 1$ or $-32,768$ to $32,767$. Such integers are called *two’s complement* numbers. Chapter 3 shows how we would encode `addi $t0,$t0,-1` or `lw $t0,-4($s0)`, which require negative numbers in the constant field of the immediate format.

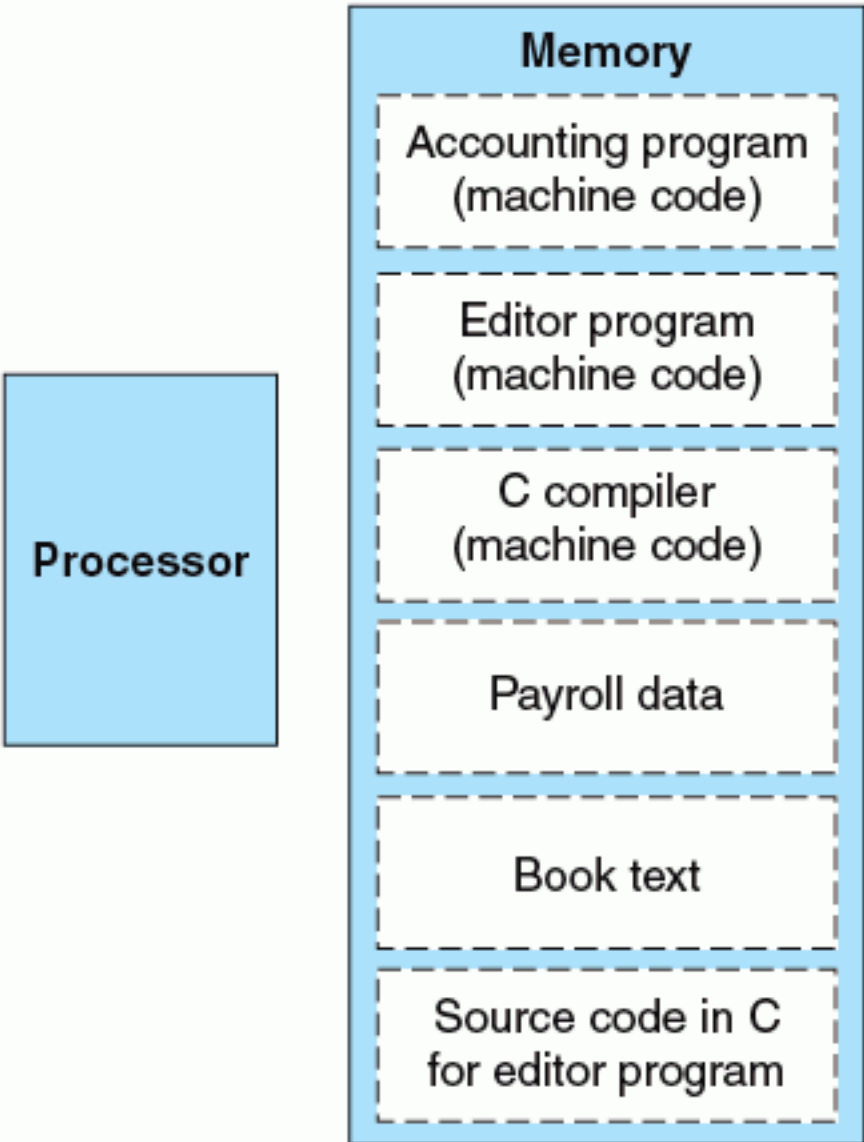


FIGURE 2.8 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

“Contrariwise,” continued Tweedledee, “if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.”
Lewis Carroll, *Alice’s Adventures in Wonderland*, 1865

2.5

Logical Operations

Although the first computers concentrated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which are stored as 8 bits, is one example of such an operation. It follows that instructions were added to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.9 shows logical operations in C and Java.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.9 C and Java logical operators and their corresponding MIPS instructions.

The first class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register \$s0 contained

0000 0000 0000 00000 000 0000 0000 0000 1001_{two} = 9_{ten}

and the instruction to shift left by 4 was executed, the new value would look like this:

0000 0000 0000 0000 0000 0000 0000 1001 0000_{two} = 144_{ten}

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (sll) and *shift right logical* (srl). The following instruction performs the operation above, assuming that the result should go in register \$t2:

sll \$t2,\$s0,4 # reg \$t2 = reg \$s0 << 4 bits

We delayed explaining the *shamt* field in the R-format. It stands for *shift amount* and is used in shift instructions. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of sll is 0 in both the op and funct fields, rd contains \$t2, rt contains \$s0, and shamt contains 4. The rs field is unused, and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by i bits gives the same result as multiplying by 2^i (Chapter 3 explains why). For example, the above sll shifts by 4, which gives the same result as multiplying by 2^4 or 16.

The first bit pattern above represents 9, and $9 \times 16 = 144$, the value of the second bit pattern.

Another useful operation that isolates fields is *AND*. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register \$t2 still contains

0000 0000 0000 0000 0000 1101 0000 0000_{two}

and register \$t1 contains

0000 0000 0000 0000 0011 1100 0000 0000_{two}

then, after executing the MIPS instruction

and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

the value of register \$t0 would be

0000 0000 0000 0000 0000 1100 0000 0000_{two}

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask “conceals” some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called *OR*. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers \$t1 and \$t2 are unchanged from the preceding example, the result of the MIPS instruction

or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

is this value in register \$t0:

0000 0000 0000 0000 0011 1101 0000 0000_{two}

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

NOR A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands.

The final logical operation is a contrarian. **NOT** takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. In keeping with the two-operand format, the designers of MIPS decided to include the instruction **NOR** (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT. For example, $A \text{ NOR } 0 = \text{NOT } (A \text{ OR } 0) = \text{NOT } (A)$.

If the register \$t1 is unchanged from the preceding example and register \$t3 has the value 0, the result of the MIPS instruction

nor \$t0,\$t1,\$t3 # reg \$t0 = ~ (reg \$t1 | reg \$t3)

is this value in register \$t0:

1111 1111 1111 1111 1100 0011 1111 1111_{two}

Figure 2.9 above shows the relationship between the C and Java operators and the MIPS instructions. Constants are useful in AND and OR logical operations as well as in arithmetic operations, so MIPS also provides the instructions *and immediate* (andi) and *or immediate* (ori). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the hardware has no immediate version. Figure 2.10, which summarizes the MIPS instructions seen thus far, highlights the logical instructions.

MIPS operands				
Name	Example	Comments		
32 registers	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers \$s0-\$s7 map to 16-23 and \$t0-\$t7 map to 8-15.		
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.		

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	\$s1 = \$s2 + 100	+ constant; overflow detected
Logical	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	\$s1 = \$s2 100	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
Data transfer	load word	lw \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Word from memory to register
	store word	sw \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Word from register to memory

FIGURE 2.10 MIPS architecture revealed thus far. Color indicates the portions introduced since Figure 2.7 on page 67. The back endpapers of this book also list the MIPS machine language.

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation. When the iteration is completed a different sequence of [instructions] is to be followed, so we must, in most cases, give two parallel trains of [instructions] preceded by an instruction as to which routine is to be followed. This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Burks, Goldstine, and von Neumann, 1947

EXAMPLE

ANSWER

2.6

Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
beq register1, register2, L1
```

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2. The mnemonic bne stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

Compiling *if-then-else* into Conditional Branches

In the following code segment, *f*, *g*, *h*, *i*, and *j* are variables. If the five variables *f* through *j* correspond to the five registers \$s0 through \$s4, what is the compiled MIPS code for this C *if* statement?

```
if (i == j) f = g + h; else f = g - h;
```

Figure 2.11 is a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want beq. In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label Else is defined below):.

```
bne $s3,$s4,Else    # go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as *j* (the label *Exit* is defined below).

```
j Exit    # go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label *Else* to this instruction. We also show the label *Exit* that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.10).

conditional branch An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

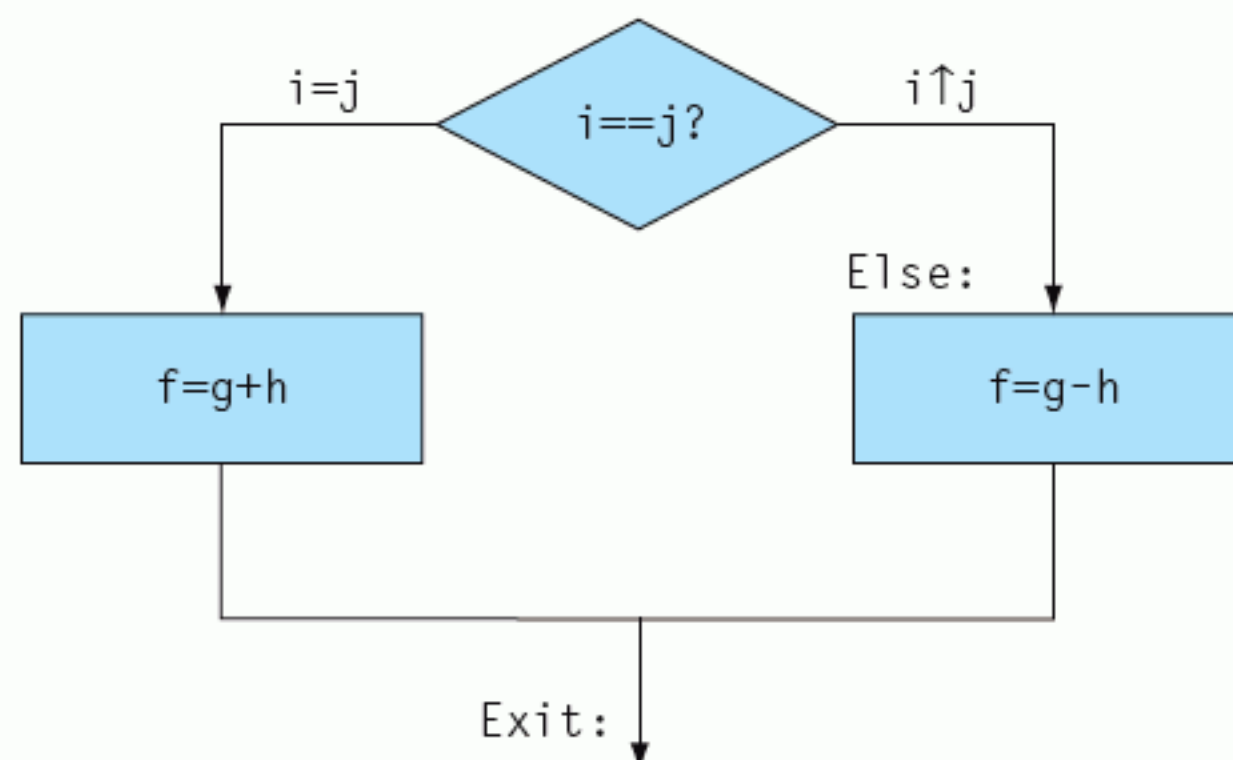


FIGURE 2.11 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Hardware Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

EXAMPLE

Compiling a *while* Loop in C

Here is a traditional loop in C:

```
while (save[i] == k)
    i += 1;
```

Assume that *i* and *k* correspond to registers *\$s3* and *\$s5* and the base of the array *save* is in *\$s6*. What is the MIPS assembly code corresponding to this C segment?

ANSWER

The first step is to load *save[i]* into a temporary register. Before we can load *save[i]* into a temporary register, we need to have its address. Before we can add *i* to the base of array *save* to form the address, we must multiply the index *i* by 4 due to the byte addressing problem. Fortunately, we can use shift left logical since shifting left by 2 bits multiplies by 4 (see page 69 in Section 2.5). We need to add the label *Loop* to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll    $t1,$s3,2    # Temp reg $t1 = 4 * i
```

To get the address of *save[i]*, we need to add *\$t1* and the base of *save* in *\$s6*:

```
add $t1,$t1,$s6    # $t1 = address of save[i]
```

Now we can use that address to load *save[i]* into a temporary register:

```
lw  $t0,0($t1)    # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:

```
bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to `i`:

```
add $s3,$s3,1 # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
      j      Loop      # go to Loop
Exit:
```

(See Exercise 2.33 for an optimization of this sequence.)

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks

Hardware Software Interface

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than*, or `slt`. For example,

```
slt    $t0, $s3, $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons. Since register `$zero` always has 0, we can already compare to 0. To compare to other values, there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
slti   $t0,$s2,10 # $t0 = 1 if $s2 < 10
```

Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

Hardware Software Interface

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal. (As you might expect, register `$zero` maps to register 0.)

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

jump address table Also called jump table. A table of addresses of alternative instruction sequences.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. See the In More Depth exercises in Section 2.20 for more details on jump address tables.

To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. The program loads the appropriate entry from the jump table into a register, and then it jumps to the proper address using a jump register. This instruction is described in Section 2.7.

Hardware Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the next lower level is the conditional branch.

Figure 2.12 summarizes the portions of MIPS assembly language described in this section, and Figure 2.13 summarizes the corresponding MIPS machine language. This step along the evolution of the MIPS language has added branches and jumps to our symbolic representation, and fixes the useful value 0 permanently in a register.

Elaboration: If you have heard about *delayed branches*, covered in Chapter 6, don't worry: The MIPS assembler makes them invisible to the assembly language programmer.

MIPS operands

Name	Example	Comments
32 registers	\$s0, \$s1, ..., \$s7 \$t0, \$t1, ..., \$t7, \$zero	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers \$s0–\$s7 map to 16–23 and \$t0–\$t7 map to 8–15. MIPS register \$zero always equals 0.
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3	Three operands; data in registers
Data transfer	load word	lw \$s1, 100(\$s2)	\$s1 = Memory[\$s2 + 100]	Data from memory to register
	store word	sw \$s1, 100(\$s2)	Memory[\$s2 + 100] = \$s1	Data from register to memory
Logical	and	and \$s1, \$s2, \$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1, \$s2, \$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1, \$s2, \$s3	\$s1 = ~(\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1, \$s2, 100	\$s1 = \$s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori \$s1, \$s2, 100	\$s1 = \$s2 100	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1, \$s2, 10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1, \$s2, 10	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq \$s1, \$s2, L	if (\$s1 == \$s2) go to L	Equal test and branch
	branch on not equal	bne \$s1, \$s2, L	if (\$s1 != \$s2) go to L	Not equal test and branch
	set on less than	slt \$s1, \$s2, \$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	slt \$s1, \$s2, 100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address

FIGURE 2.12 MIPS architecture revealed through Section 2.6. Highlighted portions show MIPS structures introduced in Section 2.6.

C has many statements for decisions and loops while MIPS has few. Which of the following do or do not explain this imbalance? Why?

Check Yourself

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
and	R	0	18	19	17	0	36	and \$s1,\$s2,\$s3
or	R	0	18	19	17	0	37	or \$s1,\$s2,\$s3
nor	R	0	18	19	17	0	39	nor \$s1,\$s2,\$s3
andi	I	12	18	17	100			andi \$s1,\$s2,100
ori	I	13	18	17	100			ori \$s1,\$s2,100
sll	R	0	0	18	17	10	0	sll \$s1,\$s2,10
srl	R	0	0	18	17	10	2	srl \$s1,\$s2,10
beq	I	4	17	18	25			beq \$s1,\$s2,100
bne	I	5	17	18	25			bne \$s1,\$s2,100
slt	R	0	18	19	17	0	42	slt \$s1,\$s2,\$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 2.13 MIPS machine language revealed through Section 2.6. Highlighted portions show MIPS structures introduced in Section 2.6. The J-format, used for jump instructions, is explained in Section 2.9. Section 2.9 also explains the proper values in address fields of branch instructions.

3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't?

1. Logical operations AND and OR implement & and | while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.7

Supporting Procedures in Computer Hardware

A **procedure** or function is one tool C or Java programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time, with parameters acting as a barrier between the procedure and the rest of the program and data, allowing it to be passed values and return results. We describe the equivalent in Java at the end of this section, but Java needs everything from a computer that C needs.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a “need to know” basis, so the spy can’t make assumptions about his employer.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Place parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention in allocating its 32 registers for procedure calling:

- `$a0–$a3`: four argument registers in which to pass parameters
- `$v0–$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register `$ra`. The **jump-and-link instruction** (`j al`) is simply written

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

program counter (PC) The register containing the address of the instruction in the program being executed

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

stack A data structure for spilling registers organized as a last-in-first-out queue.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.

```
jal ProcedureAddress
```

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This “link,” stored in register `$ra`, is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *PC* in the MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction saves `PC + 4` in register `$ra` to link to the following instruction to set up the procedure return.

To support such situations, computers like MIPS use a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register:

```
jr $ra
```

The jump register instruction jumps to the address stored in register `$ra`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `$a0–$a3` and uses `jal X` to jump to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in `$v0–$v1`, and returns control to the caller using `jr $ra`.

Using More Registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the Hardware Software Interface section on page 58.

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The **stack pointer** is adjusted by one word for each register that is saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a *push*, and removing data from the stack is called a *pop*.

MIPS software allocates another register just for the stack: the stack pointer (`$sp`), used to save the registers needed by the callee. By historical precedent, stacks “grow” from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a C Procedure That Doesn't Call Another Procedure

Let's turn the example on page 51 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

What is the compiled MIPS assembly code?

The parameter variables g, h, i, and j correspond to the argument registers \$a0, \$a1, \$a2, and \$a3, and f corresponds to \$s0. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 51, which uses two temporary registers. Thus, we need to save three registers: \$s0, \$t0, and \$t1. We “push” the old values onto the stack by creating space for three words on the stack and then store them:

```
addi $sp,$sp,-12 # adjust stack to make room for 3 items
sw   $t1, 8($sp) # save register $t1 for use afterwards
sw   $t0, 4($sp) # save register $t0 for use afterwards
sw   $s0, 0($sp) # save register $s0 for use afterwards
```

Figure 2.14 shows the stack before, during, and after the procedure call. The next three statements correspond to the body of the procedure, which follows the example on page 51:

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

To return the value of f, we copy it into a return value register:

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

EXAMPLE**ANSWER**

Before returning, we restore the three old values of the registers we saved by “popping” them from the stack:

```
lw  $s0, 0($sp) # restore register $s0 for caller
lw  $t0, 4($sp) # restore register $t0 for caller
lw  $t1, 8($sp) # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr  $ra    # jump back to calling routine
```

In the example above we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- \$t0–\$t9: 10 temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- \$s0–\$s7: 8 saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller (procedure doing the calling) does not expect registers \$t0 and \$t1 to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore \$s0, since the callee must assume that the caller needs its value.

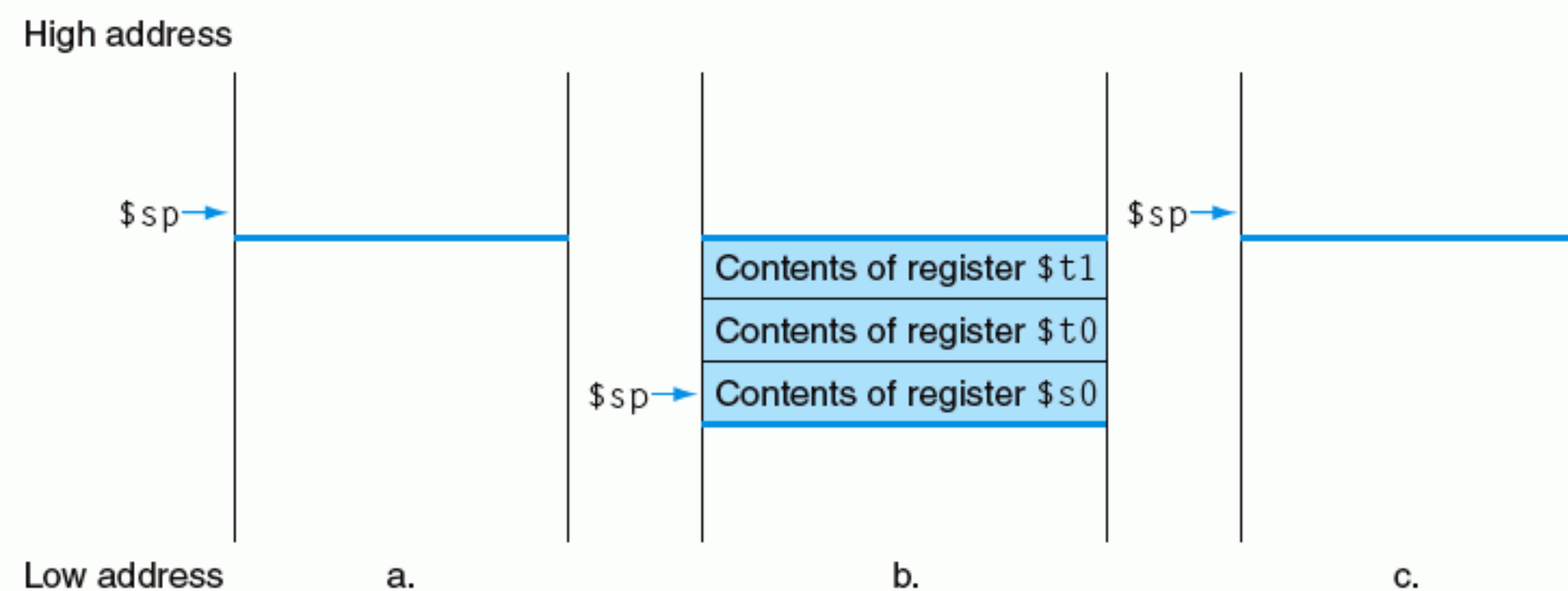


FIGURE 2.14 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

Nested Procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register \$a0 and then using `jal A`. Then suppose that procedure A calls procedure B via `jal B` with an argument of 7, also placed in \$a0. Since A hasn't finished its task yet, there is a conflict over the use of register \$a0. Similarly, there is a conflict over the return address in register \$ra, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved onto the stack, just as we did with the saved registers. The caller pushes any argument registers (\$a0–\$a3) or temporary registers (\$t0–\$t9) that are needed after the call. The callee pushes the return address register \$ra and any saved registers (\$s0–\$s7) used by the callee. The stack pointer \$sp is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

What is the MIPS assembly code?

EXAMPLE

ANSWER

The parameter variable `n` corresponds to the argument register `$a0`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `$a0`:

```
fact:
    addi    $sp,$sp,-8    # adjust stack for 2 items
    sw      $ra, 4($sp)   # save the return address
    sw      $a0, 0($sp)   # save the argument n
```

The first time `fact` is called, `sw` saves an address in the program that called `fact`. The next two instructions test if `n` is less than 1, going to `L1` if $n \geq 1$.

```
    slti    $t0,$a0,1     # test for  $n < 1$ 
    beq      $t0,$zero,L1  # if  $n \geq 1$ , go to L1
```

If `n` is less than 1, `fact` returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in `$v0`. It then pops the two saved values off the stack and jumps to the return address:

```
    addi    $v0,$zero,1   # return 1
    addi    $sp,$sp,8      # pop 2 items off stack
    jr      $ra            # return to after jal
```

Before popping two items off the stack, we could have loaded `$a0` and `$ra`. Since `$a0` and `$ra` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: addi    $a0,$a0,-1    #  $n \geq 1$ : argument gets  $(n - 1)$ 
    jalfact                # call fact with  $(n - 1)$ 
```

The next instruction is where `fact` returns. Now the old return address and old argument are restored, along with the stack pointer:

```
    lw      $a0, 0($sp)   # return from jal:restore argument n
    lw      $ra, 4($sp)   # restore the return address
    addi    $sp, $sp,8     # adjust stack pointer to pop 2 items
```

Next, the value register \$v0 gets the product of old argument \$a0 and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul    $v0,$a0,$v0    # return n * fact (n - 1)
```

Finally, fact jumps again to the return address:

```
jr     $ra             # return to the caller
```

A C variable is a location in storage, and its interpretation depends both on its *type* and *storage class*. Types are discussed in detail in Chapter 3, but examples include integers and characters. C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword `static`. The rest are automatic. To simplify access to static data, MIPS software reserves another register, called the **global pointer**, or \$gp.

Hardware
Software
Interface

global pointer The register that is reserved to point to static data.

Figure 2.15 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack. The stack above \$sp is preserved simply by making sure the callee does not write above \$sp; \$sp is itself preserved by the callee adding exactly the same amount that was subtracted from it, and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there. These actions also guarantee that the caller will get the same data back on a load from the stack as it put into the stack on a store because the callee promises to preserve \$sp and because the callee also promises not to modify the caller’s portion of the stack, that is, the area above the \$sp at the time of the call.

Preserved	Not preserved
Saved registers: \$s0–\$s7	Temporary registers: \$t0–\$t9
Stack pointer register: \$sp	Argument registers: \$a0–\$a3
Return address register: \$ra	Return value registers: \$v0–\$v1
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.15 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following sections, they are also preserved.

procedure frame Also called **activation record**. The segment of the stack containing a procedure’s saved registers and local variables.

frame pointer A value denoting the location of the saved registers and local variables for a given procedure.

Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure that do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure’s saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.16 shows the state of the stack before, during, and after the procedure call.

Some MIPS software uses a **frame pointer** (\$fp) to point to the first word of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We’ve been avoiding \$fp by avoiding changes to \$sp within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.

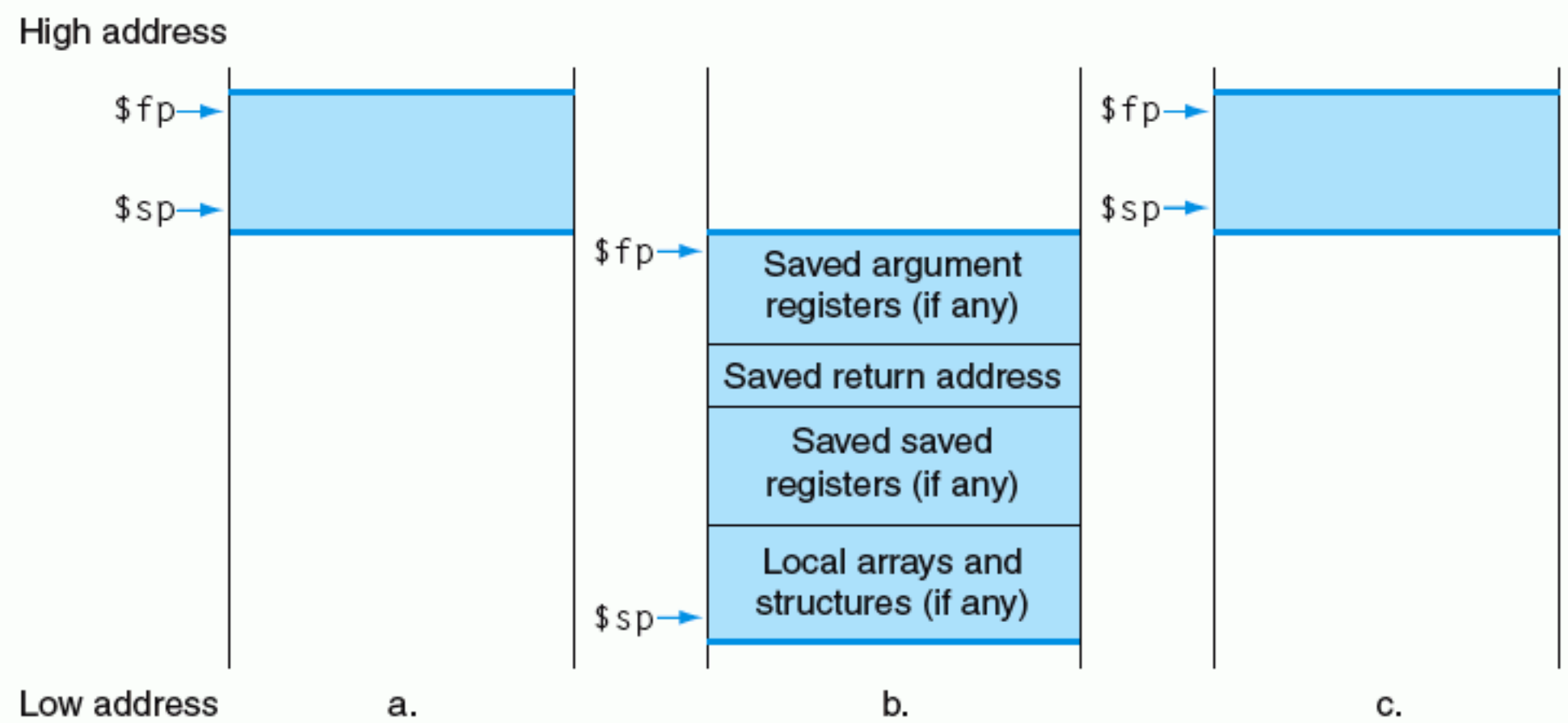


FIGURE 2.16 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (\$fp) points to the first word of the frame, often a saved argument register, and the stack pointer (\$sp) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it’s easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in \$sp on a call, and \$sp is restored using \$fp.

Allocating Space for New Data on the Heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. Figure 2.17 shows the MIPS convention for allocation of memory. The stack starts in the high end of memory and grows down. The first part of the low end of memory is reserved, followed by the home of the MIPS machine code, traditionally called the **text segment**. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be to a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap*, and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

C allocates and frees space on the heap with explicit functions. `malloc()` allocates space on the heap and returns a pointer to it, and `free()` releases space on the stack to which the pointer points. Memory allocation is controlled by programs in C, and it is the source of many common and difficult bugs. Forgetting to free space

text segment The segment of a Unix object file that contains the machine language code for routines in the source file.

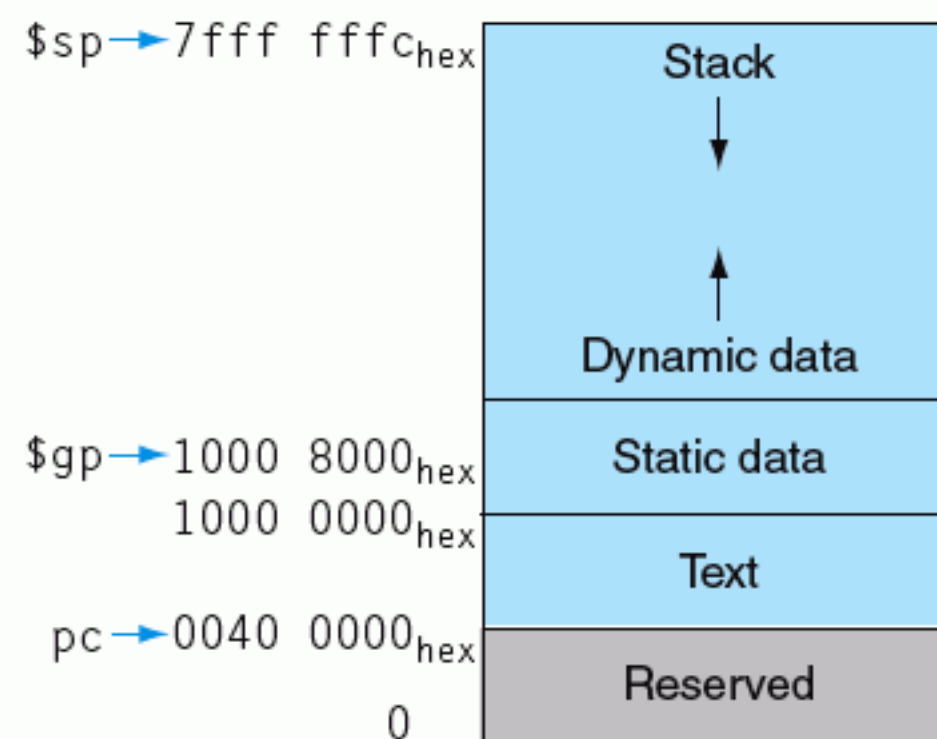


FIGURE 2.17 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. Starting top down, the stack pointer is initialized to 7fff fffc_{hex} and grows down toward the data segment. At the other end, the program code (“text”) starts at 0040 0000_{hex}. The static data starts at 1000 0000_{hex}. Dynamic data, allocated by `malloc` in C and via `new` in Java, is next and grows up toward the stack in an area called the heap. The global pointer, `$gp`, is set to an address to make it easy to access data. It is initialized to 1000 8000_{hex} so that it can access from 1000 0000_{hex} to 1000 ffff_{hex} using the positive and negative 16-bit offsets from `$gp` (see two’s complement addressing in Chapter 3).

leads to a “memory leak” which eventually uses up so much memory that the operating system may crash. Freeing space too early leads to a “dangling pointers,” which can cause pointers to point to things that the program never intended.

Figure 2.18 summarizes the register conventions for the MIPS assembly language. Figures 2.19 and 2.20 summarize the parts of the MIPS assembly instructions described so far and the corresponding MIPS machine instructions.

Elaboration: What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first four parameters to be in registers \$a0 through \$a3 and the rest in memory, addressable via the frame pointer.

As mentioned in the caption of Figure 2.16, the frame pointer is convenient because all references to variables in the stack within a procedure will have the same offset. The frame pointer is not necessary, however. The GNU MIPS C compiler uses a frame pointer, but the C compiler from MIPS/Silicon Graphics does not; it uses register 30 as another save register (\$s8).

jal actually saves the address of the instruction that follows jal into register \$ra, thereby allowing a procedure return to be simply jr \$ra.

Check Yourself

Which of the following statements about C and Java are generally true?

- 1. Procedure calls in C are faster than method invocation in Java.
- 2. C programmers manage data explicitly while it’s automatic in Java.
- 3. C leads to more pointer bugs and memory leak bugs than does Java.
- 4. C passes parameters in registers while Java passes them on the stack.

Name	Register number	Usage	Preserved on call?
\$zero	0	the constant value 0	n.a.
\$v0–\$v1	2–3	values for results and expression evaluation	no
\$a0–\$a3	4–7	arguments	no
\$t0–\$t7	8–15	temporaries	no
\$s0–\$s7	16–23	saved	yes
\$t8–\$t9	24–25	more temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

FIGURE 2.18 MIPS register conventions. Register 1, called \$at, is reserved for the assembler (see Section 2.10), and registers 26–27, called \$k0–\$k1, are reserved for the operating system.

MIPS operands

Name	Example	Comments
32 registers	<code>\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register <code>\$zero</code> always equals 0. <code>\$gp</code> (28) is the global pointer, <code>\$sp</code> (29) is the stack pointer, <code>\$fp</code> (30) is the frame pointer, and <code>\$ra</code> (31) is the return address.
2^{30} memory words	<code>Memory[0], Memory[4], . . . , Memory[4294967292]</code>	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	<code>add \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 + \$s3</code>	three register operands
	subtract	<code>sub \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 - \$s3</code>	three register operands
Data transfer	load word	<code>lw \$s1,100(\$s2)</code>	<code>\$s1 = Memory[\$s2 + 100]</code>	Data from memory to register
	store word	<code>sw \$s1,100(\$s2)</code>	<code>Memory[\$s2 + 100] = \$s1</code>	Data from register to memory
Logical	and	<code>and \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 & \$s3</code>	three reg. operands; bit-by-bit AND
	or	<code>or \$s1,\$s2,\$s3</code>	<code>\$s1 = \$s2 \$s3</code>	three reg. operands; bit-by-bit OR
	nor	<code>nor \$s1,\$s2,\$s3</code>	<code>\$s1 = ~ (\$s2 \$s3)</code>	three reg. operands; bit-by-bit NOR
	and immediate	<code>andi \$s1,\$s2,100</code>	<code>\$s1 = \$s2 & 100</code>	Bit-by-bit AND reg with constant
	or immediate	<code>ori \$s1,\$s2,100</code>	<code>\$s1 = \$s2 100</code>	Bit-by-bit OR reg with constant
	shift left logical	<code>sll \$s1,\$s2,10</code>	<code>\$s1 = \$s2 << 10</code>	Shift left by constant
	shift right logical	<code>srl \$s1,\$s2,10</code>	<code>\$s1 = \$s2 >> 10</code>	Shift right by constant
Conditional branch	branch on equal	<code>beq \$s1,\$s2,L</code>	if (<code>\$s1 == \$s2</code>) go to L	Equal test and branch
	branch on not equal	<code>bne \$s1,\$s2,L</code>	if (<code>\$s1 != \$s2</code>) go to L	Not equal test and branch
	set on less than	<code>slt \$s1,\$s2,\$s3</code>	if (<code>\$s2 < \$s3</code>) <code>\$s1 = 1</code> ; else <code>\$s1 = 0</code>	Compare less than; used with <code>beq</code> , <code>bne</code>
	set on less than immediate	<code>slt \$s1,\$s2,100</code>	if (<code>\$s2 < 100</code>) <code>\$s1 = 1</code> ; else <code>\$s1 = 0</code>	Compare less than immediate; used with <code>beq</code> , <code>bne</code>
Unconditional jump	jump	<code>j L</code>	go to L	Jump to target address
	jump register	<code>jr \$ra</code>	go to <code>\$ra</code>	For procedure return
	jump and link	<code>jal L</code>	<code>\$ra = PC + 4</code> ; go to L	For procedure call

FIGURE 2.19 MIPS architecture revealed through Section 2.7. Highlighted portions show MIPS assembly language structures introduced in Section 2.7. The J-format, used for jump and jump-and-link instructions, is explained in Section 2.9.

MIPS machine language								
Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
and	R	0	18	19	17	0	36	and \$s1,\$s2,\$s3
or	R	0	18	19	17	0	37	or \$s1,\$s2,\$s3
nor	R	0	18	19	17	0	39	nor \$s1,\$s2,\$s3
andi	I	12	18	17	100			andi \$s1,\$s2,100
ori	I	13	18	17	100			ori \$s1,\$s2,100
sll	R	0	0	18	17	10	0	sll \$s1,\$s2,10
srl	R	0	0	18	17	10	2	srl \$s1,\$s2,10
beq	I	4	17	18	25			beq \$s1,\$s2,100
bne	I	5	17	18	25			bne \$s1,\$s2,100
slt	R	0	18	19	17	0	42	slt \$s1,\$s2,\$s3
j	J	2	2500					j 10000 (see Section 2.9)
jr	R	0	31	0	0	0	8	jr \$ra
jal	J	3	2500					jal 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 2.20 MIPS machine language revealed through Section 2.7. Highlighted portions show MIPS assembly language structures introduced in Section 2.7. The J-format, used for jump and jump-and-link instructions, is explained in Section 2.9. This section also explains why putting 25 in the address field of beq and bne machine language instructions is equivalent to 100 in assembly language.

!(@ | = >
(wow open tab at bar is
great)
Fourth line of the keyboard
poem “Hatless Atlas,” 1991
(some give names to ASCII
characters: “!” is “wow,”“(“ is
open, “|” is bar, and so on)

2.8

Communicating with People

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today use 8-bit bytes to represent characters, with the American Standard Code for Information Interchange (ASCII) being the representation that nearly everyone follows. Figure 2.21 summarizes ASCII.

A series of instructions can extract a byte from a word, so load word and store word are sufficient for transferring bytes as well as words. Because of the popularity

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.21 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string.

of text in some programs, however, MIPS provides instructions to move bytes. Load byte (`lb`) loads a byte from memory, placing it in the rightmost 8 bits of a register. Store byte (`sb`) takes a byte from the rightmost 8 bits of a register and writes it to memory. Thus, we copy a byte with the sequence

```
lb $t0,0($sp)           # Read byte from source
sb $t0,0($gp)           # Write byte to destination
```

Characters are normally combined into strings, which have a variable number of characters. There are three choices for representing a string: (1) the first position of the string is reserved to give the length of a string, (2) an accompanying variable has the length of the string (as in a structure), or (3) the last position of a string is indicated by a character used to mark the end of a string. C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII). Thus, the string “Cal” is represented in C by the following 4 bytes, shown as decimal numbers: 67, 97, 108, 0.

EXAMPLE**Compiling a String Copy Procedure, Showing How to Use C Strings**

The procedure `strcpy` copies string `y` to string `x` using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```

What is the MIPS assembly code?

ANSWER

Below is the basic MIPS assembly code segment. Assume that base addresses for arrays `x` and `y` are found in `$a0` and `$a1`, while `i` is in `$s0`. `strcpy` adjusts the stack pointer and then saves the saved register `$s0` on the stack:

```
strcpy:
    addi    $sp,$sp,-4    # adjust stack for 1 more item
    sw      $s0, 0($sp)  # save $s0
```

To initialize `i` to 0, the next instruction sets `$s0` to 0 by adding 0 to 0 and placing that sum in `$s0`:

```
add    $s0,$zero,$zero    # i = 0 + 0
```

This is the beginning of the loop. The address of `y[i]` is first formed by adding `i` to `y[]`:

```
L1: add    $t1,$s0,$a1    # address of y[i] in $t1
```

Note that we don't have to multiply `i` by 4 since `y` is an array of *bytes* and not of words, as in prior examples.

To load the character in `y[i]`, we use load byte, which puts the character into `$t2`:

```
lb      $t2, 0($t1)    # $t2 = y[i]
```

A similar address calculation puts the address of $x[i]$ in $\$t3$, and then the character in $\$t2$ is stored at that address.

```
add    $t3,$s0,$a0    # address of x[i] in $t3
sb     $t2, 0($t3)    # x[i] = y[i]
```

Next we exit the loop if the character was 0; that is, if it is the last character of the string:

```
beq    $t2,$zero,L2   # if y[i] == 0, go to L2
```

If not, we increment i and loop back:

```
addi   $s0, $s0,1     # i = i + 1
j      L1              # go to L1
```

If we don't loop back, it was the last character of the string; we restore $\$s0$ and the stack pointer, and then return.

```
L2: lw    $s0, 0($sp)  # y[i] == 0: end of string;
                        # restore old $s0
addi    $sp,$sp,4      # pop 1 word off stack
jr      $ra            # return
```

String copies usually use pointers instead of arrays in C to avoid the operations on i in the code above. See Section 2.15 for an explanation of arrays versus pointers.

Since the procedure `strcpy` above is a leaf procedure, the compiler could allocate i to a temporary register and avoid saving and restoring $\$s0$. Hence, instead of thinking of the $\$t$ registers as being just for temporaries, we can think of them as registers that the callee should use whenever convenient. When a compiler finds a leaf procedure, it exhausts all temporary registers before using registers it must save.

Characters and Strings in Java

Unicode is a universal encoding of the alphabets of most human languages. Figure 2.22 is a list of Unicode alphabets; there are about as many *alphabets* in Unicode as there are useful *symbols* in ASCII. To be more inclusive, Java uses Unicode for characters. By default, it uses 16 bits to represent a character.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURE 2.22 Example alphabets in Unicode. Unicode version 4.0 has more than 160 “blocks,” which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as 8 bits and uses 16–32 bits for the other characters. UTF-32 uses 32 bits per character. To learn more, see www.unicode.org.

The MIPS instruction set has explicit instructions to load and store such 16-bit quantities, called *halfwords*. Load half (lh) loads a halfword from memory, placing it in the rightmost 16 bits of a register. Store half (sh) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence

```
lh $t0,0($sp)  # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

Strings are a standard Java class with special built-in support and predefined methods for concatenation, comparison, and conversion. Unlike C, Java includes a word that gives the length of the string, similar to Java arrays.

Elaboration: MIPS software tries to keep the stack aligned to word addresses, allowing the program to always use lw and sw (which must be aligned) to access the stack. This convention means that a char variable allocated on the stack occupies 4 bytes, even though it needs less. However, a C string variable or an array of bytes will pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

Which of the following statements about characters and strings in C and Java are true?

Check Yourself

- 1. A string in C takes about half the memory as the same string in Java.
- 2. Strings are just an informal name for single-dimension arrays of characters in C and Java.
- 3. Strings in C and Java use null (0) to mark the end of a string.
- 4. Operations on strings, like length, are faster in C than in Java.

2.9

MIPS Addressing for 32-Bit Immediates and Addresses

Although keeping all MIPS instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches and jumps.

32-Bit Immediate Operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (*lui*) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. Figure 2.23 shows the operation of *lui*.

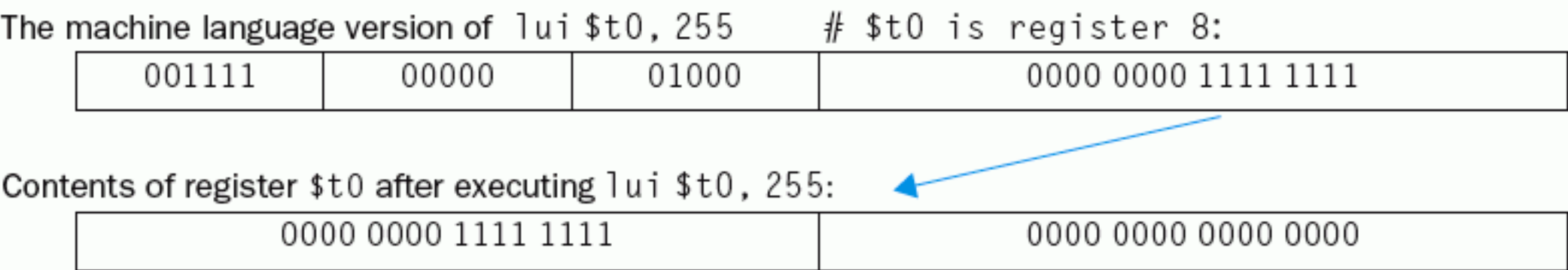


FIGURE 2.23 The effect of the *lui* instruction. The instruction *lui* transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

Hardware Software Interface

Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions. If this job falls to the assembler, as it does for MIPS software, then the assembler must have a temporary register available in which to create the long values. This is a reason for the register `$at`, which is reserved for the assembler.

Hence, the symbolic representation of the MIPS machine language is no longer limited by the hardware, but to whatever the creator of an assembler chooses to include (see Section 2.10). We stick close to the hardware to explain the architecture of the computer, noting when we use the enhanced language of the assembler that is not found in the processor.

EXAMPLE

ANSWER

Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61    # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `$s0` afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to add the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register `$s0` is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Elaboration: Creating 32-bit constants needs care. The instruction `addi` copies the leftmost bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. *Logical or immediate* from Section 2.5 loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with `lui` to create 32-bit constants.

Addressing in Branches and Jumps

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
j    10000    # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see on the next page):

2	10000
6 bits	26 bits

where the value of the jump opcode is 2 and the jump address is 10000.

Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

```
bne  $s0,$s1,Exit    # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^{16} , which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch address}$$

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. The question is then, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC2000 benchmarks go to locations less than 16 instructions away. Since the program counter (PC) contains the address of the current instruction, we can

PC-relative addressing An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

branch within $\pm 2^{15}$ words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if* statements are much smaller than 2^{16} words, so the PC is the ideal choice.

This form of branch addressing is called **PC-relative addressing**. As we shall see in Chapter 5, it is convenient for the hardware to increment the PC early to point to the next instruction. Hence, the MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to the current instruction (PC).

Like most recent computers, MIPS uses PC-relative addressing for all conditional branches because the destination of these instructions is likely to be close to the branch. On the other hand, jump-and-link instructions invoke procedures that have no reason to be near the call, and so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 16-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address. Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.

Elaboration: Since the PC is 32 bits, 4 bits must come from somewhere else. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker (Section 2.9) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

EXAMPLE

Showing Branch Offset in Machine Language

The *while* loop on page 74 was compiled into this MIPS assembler code:

```
Loop: sll    $t1,$s3,2    # Temp reg $t1 = 4 * i
      add $t1,$t1,$s6    # $t1 = address of save[i]
      lw   $t0,0($t1)    # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3,$s3,1     # i = i + 1
      j    Loop          # go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

The assembled instructions and their addresses would look like this:

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word. The `bne` instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction ($8 + 80016$) instead of relative to the branch instruction ($12 + 80012$) or using the full destination address (80024). The jump instruction on the last line does use the full address ($20000 \times 4 = 80000$), corresponding to the label `Loop`.

ANSWER

Nearly every conditional branch is to a nearby location, but occasionally it branches far away, farther than can be represented in the 16 bits of the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.

Hardware
Software
Interface

Branching Far Away

Given a branch on register `$s0` being equal to register `$s1`,

```
beq    $s0,$s1, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

These instructions replace the short-address conditional branch:

```
      bne    $s0,$s1, L2
      j      L1
L2:
```

EXAMPLE

ANSWER

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

MIPS Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes**. The MIPS addressing modes are the following:

1. **Register addressing**, where the operand is a register
2. **Base** or **displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
3. **Immediate addressing**, where the operand is a constant within the instruction itself
4. **PC-relative addressing**, where the address is the sum of the PC and a constant in the instruction
5. **Pseudodirect addressing**, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Hardware Software Interface

Although we show the MIPS architecture as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions (see [Appendix D](#)). These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in a way that lets software move compatibly upward to the next generation of architecture.

Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (addi) and register (add) addressing. Figure 2.24 shows how operands are identified for each addressing mode. [In More Depth](#) shows other addressing modes found in the IBM PowerPC.

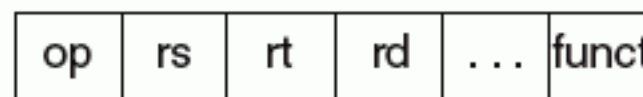
Decoding Machine Language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at a core dump. Figure 2.25 shows the MIPS encoding of the fields for the MIPS machine language. This figure helps when translating by hand between assembly language and machine language.

1. Immediate addressing



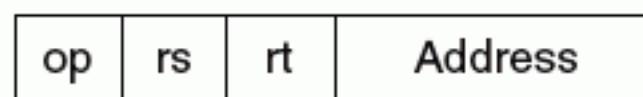
2. Register addressing



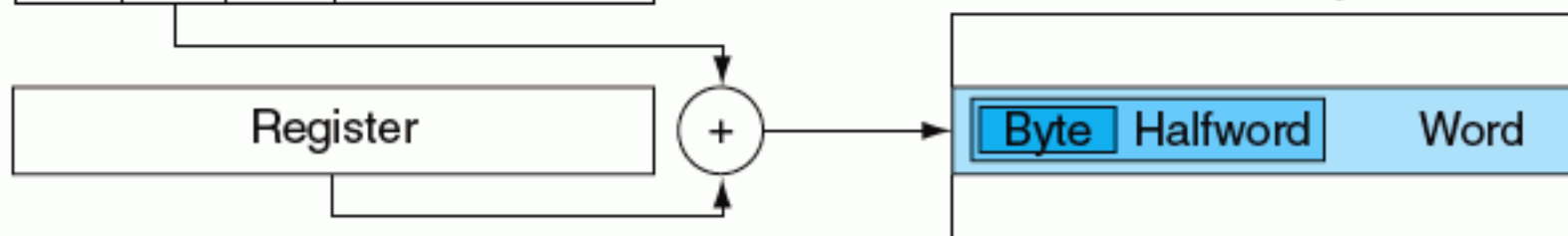
Registers

Register

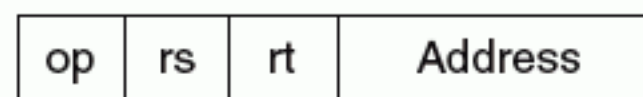
3. Base addressing



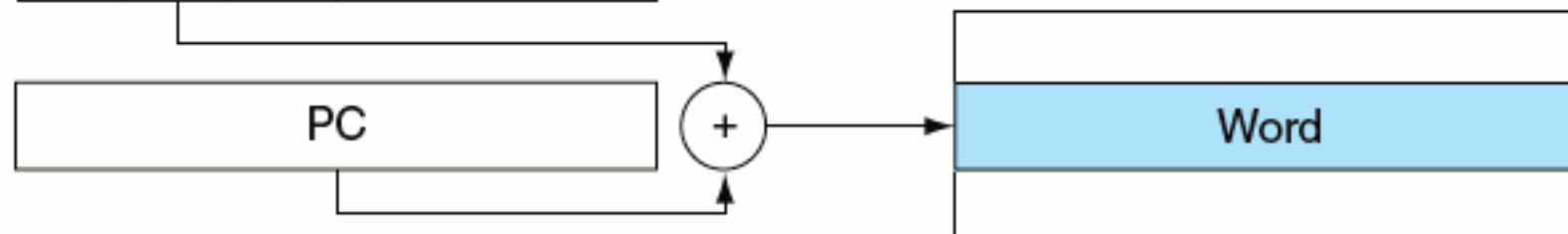
Memory



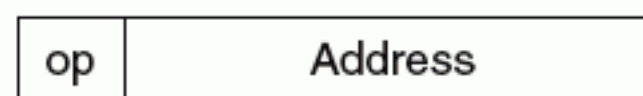
4. PC-relative addressing



Memory



5. Pseudodirect addressing



Memory

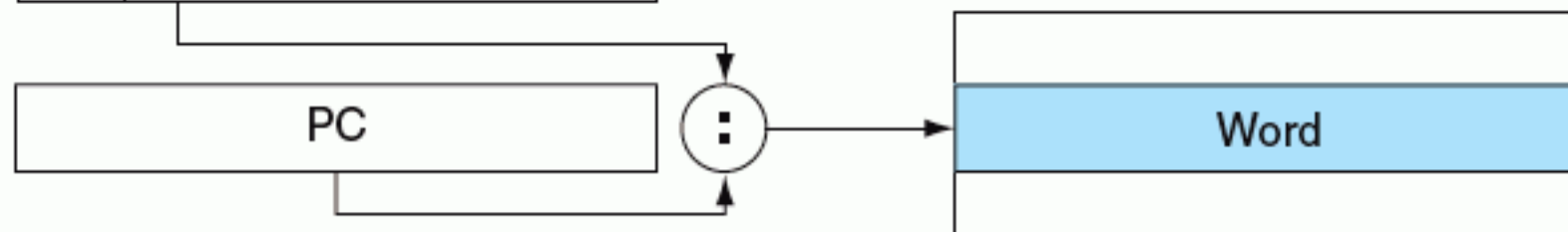


FIGURE 2.24 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

EXAMPLE

ANSWER

Decoding Machine Code

What is the assembly language statement corresponding to this machine instruction?

00af8020hex

The first step in converting hexadecimal to binary is to find the op fields:

```
(Bits: 31 28 26                                     5 2 0)
       0000 0000 1010 1111 1000 0000 0010 0000
```

We look at the `op` field to determine the operation. Referring to Figure 2.25, when bits 31–29 are 000 and bits 28–26 are 000, it is an R-format instruction. Let’s reformat the binary instruction into R-format fields, listed in Figure 2.26:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

The bottom portion of Figure 2.25 determines the operation of an R-format instruction. In this case, bits 5–3 are 100 and bits 2–0 are 000, which means this binary pattern represents an add instruction.

We decode the rest of the instruction by looking at the field values. The decimal values are 5 for the rs field, 15 for rt, 16 for rd (shamt is unused). Figure 2.18 says these numbers represent registers \$a1, \$t7, and \$s0. Now we can show the assembly instruction:

```
add $s0,$a1,$t7
```

Figure 2.26 shows all the MIPS instruction formats. Figure 2.27 shows the MIPS assembly language revealed in Chapter 2; the remaining hidden portion of MIPS instructions deals mainly with arithmetic covered in the next chapter.

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						
op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								
op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

FIGURE 2.25 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, the top portion of the figure shows **load word** in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, R-format in row 0 and column 0 (op = 000000_{two}) is defined in the bottom part of the figure. Hence, **subtract** in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010_{two} and the op field (bits 31–26) is 000000_{two}. The FlPt value in row 2, column 1 is defined in Figure 3.20 in Chapter 3. Bltz/gez is the opcode for four instructions found in [Appendix A](#): bltz, bgez, bltzal, and bgezal. Chapter 2 describes instructions given in full name using color, while Chapter 3 describes instructions given in mnemonics using color. Appendix A covers all instructions.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, <i>i</i> mm. format
J-format	op	target address					Jump instruction format

FIGURE 2.26 MIPS instruction formats in Chapter 2. Highlighted portions show instruction formats introduced in this section.

Check Yourself

What is the range of addresses for conditional branches in MIPS ($K = 1024$)?

- 1. Addresses between 0 and $64K - 1$
- 2. Addresses between 0 and $256K - 1$
- 3. Addresses up to about 32K before the branch to about 32K after
- 4. Addresses up to about 128K before the branch to about 128K after

What is the range of addresses for jump and jump and link in MIPS ($M = 1024K$)?

- 1. Addresses between 0 and $64M - 1$
- 2. Addresses between 0 and $256M - 1$
- 3. Addresses up to about 32M before the branch to about 32M after
- 4. Addresses up to about 128M before the branch to about 128M after
- 5. Anywhere within a block of 64M addresses where the PC supplies the upper 6 bits
- 6. Anywhere within a block of 256M addresses where the PC supplies the upper 4 bits

What is the MIPS assembly language instruction corresponding to the machine instruction with the value $0000\ 0000_{\text{hex}}$?

- 1. j
- 2. R-format
- 3. addi
- 4. sll
- 5. mfc0
- 6. Undefined opcode: there is no legal instruction that corresponds to 0.

MIPS operands

Name	Example	Comments
32 registers	\$s0–\$s7, \$t0–\$t9, \$zero, \$a0–\$a3, \$v0–\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants.
2 ³⁰ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 – \$s3	Three register operands
	add immediate	addi \$s1,\$s2,100	\$s1 = \$s2 + 100	Used to add constants
Data transfer	load word	lw \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Word from memory to register
	store word	sw \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Word from register to memory
	load half	lh \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Halfword memory to register
	store half	sh \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Halfword register to memory
	load byte	lb \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Byte from memory to register
	store byte	sb \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Byte from register to memory
	load upper immed.	lui \$s1,100	\$s1 = 100 * 2 ¹⁶	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	\$s1 = \$s2 100	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

FIGURE 2.27 MIPS assembly language revealed in Chapter 2. Highlighted portions show portions from Sections 2.8 and 2.9.

2.10 Translating and Starting a Program

This section describes the four steps in transforming a C program in a file on disk into a program running on a computer. Figure 2.28 shows the translation hierarchy. Some systems combine these steps to reduce translation time, but these are the logical four phases that programs go through. This section follows this translation hierarchy.

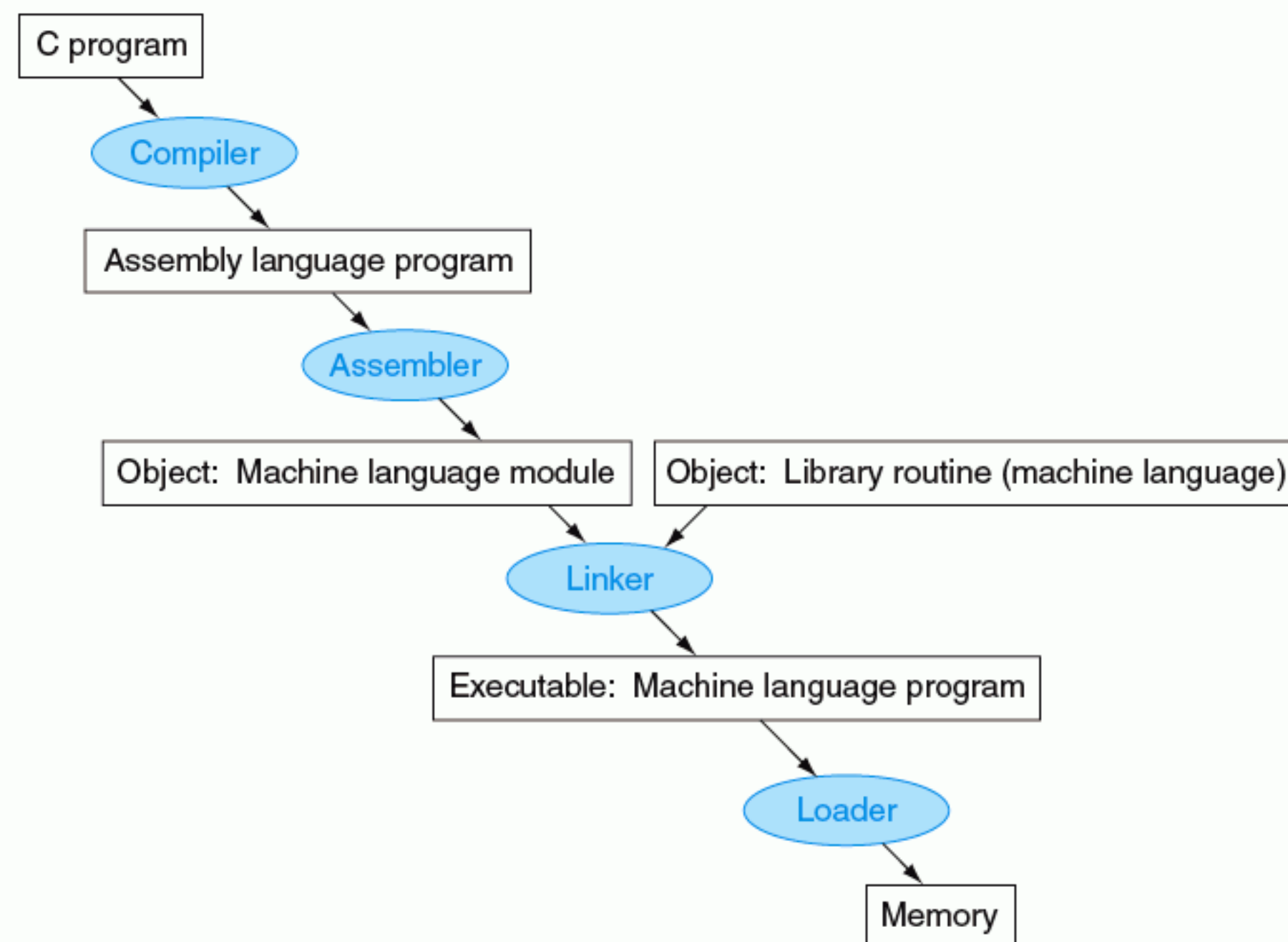


FIGURE 2.28 A translation hierarchy for C. A high-level-language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined together. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routes are `x.so`, and executable files by default are called `a.out`. MS-DOS uses the suffixes `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL`, and `.EXE` to the same effect.

Compiler

The compiler transforms the C program into an *assembly language program*, a symbolic form of what the machine understands. High-level-language programs take many fewer lines of code than assembly language, so programmer productivity is much higher.

In 1975, many operating systems and assemblers were written in **assembly language** because memories were small and compilers were inefficient. The 128,000-fold increase in memory capacity per single DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as good as an assembly language expert, and sometimes even better for large programs.

assembly language A symbolic language that can be translated into binary.

Assembler

As mentioned on page 96, since assembly language is the interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right. The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called **pseudoinstructions**.

As mentioned above, the MIPS hardware makes sure that register `$zero` always has the value 0. That is, whenever register `$zero` is used, it supplies a 0, and the programmer cannot change the value of register `$zero`. Register `$zero` is used to create the assembly language instruction `move` that copies the contents of one register to another. Thus the MIPS assembler accepts this instruction even though it is not found in the MIPS architecture:

```
move $t0,$t1      # register $t0 gets register $t1
```

The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:

```
add  $t0,$zero,$t1 # register $t0 gets 0 + register $t1
```

The MIPS assembler also converts `blt` (branch on less than) into the two instructions `slt` and `bne` mentioned in the example on page 96. Other examples include `bgt`, `bge`, and `ble`. It also converts branches to faraway locations into a branch and jump. As mentioned above, the MIPS assembler allows 32-bit constants to be loaded into a register despite the 16-bit limit of the immediate instructions.

In summary, pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. The only cost is reserving one register, `$at`, for use by the assembler. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the MIPS

pseudoinstruction A common variation of assembly language instructions often treated as if it were an instruction in its own right.

machine language Binary representation used for communication within a computer system.

symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy.

architecture and to be sure to get best performance, however, study the real MIPS instructions found in Figures 2.25 and 2.27.

Assemblers will also accept numbers in a variety of bases. In addition to binary and decimal, they usually accept a base that is more succinct than binary yet converts easily to a bit pattern. MIPS assemblers use hexadecimal,

Such features are convenient, but the primary task of an assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of **machine language** instructions, data, and information needed to place instructions properly in memory.

To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instructions in a **symbol table**. As you might expect, the table contains pairs of symbol and address.

The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use either *static data*, which is allocated throughout the program, or *dynamic data*, which can grow or shrink as needed by the program.)
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

The next subsection shows how to attach such routines that have already been assembled, such as library routines.

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to

compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a **link editor** or **linker**, which takes all the independently assembled machine language programs and “stitches” them together.

There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions, jump instructions, and data addresses, so the job of this program is much like that of an editor: It finds the old addresses and replaces them with the new addresses. Editing is the origin of the name “link editor,” or linker for short. The reason a linker makes sense is that it is much faster to patch code than it is to recompile and reassemble.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that Figure 2.17 on page 87 shows the MIPS convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module’s instructions and data will be placed relative to other modules. When the linker places a module in memory, all *absolute* references, that is, memory addresses that are not relative to a register, must be *relocated* to reflect its true location.

The linker produces an **executable file** that can be run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references. It is possible to have partially linked files, such as library routines, which still have unresolved addresses and hence result in object files.

linker Also called **link editor**. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

executable file A functional program in the format of an object file that contains no unresolved references, relocation information, symbol table, or debugging information.

Linking Object Files

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures A and B and the instructions that refer to the addresses of data words X and Y.

EXAMPLE

Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	
Object file header			
	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

ANSWER

Procedure A needs to find the address for the variable labeled X to put in the load instruction and to find the address of procedure B to place in the jal instruction. Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its jal instruction.

From Figure 2.17 on page 87, we know that the text segment starts at address 40 0000_{hex} and the data segment at 1000 0000_{hex}. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is 40 0100_{hex}, and its data starts at 1000 0020_{hex}.

Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

	0040 0100 _{hex}	sw \$a1, 8020 _{hex} (\$gp)
	0040 0104 _{hex}	jal 40 0000 _{hex}

Data segment	Address	
	1000 0000 _{hex}	(X)

	1000 0020 _{hex}	(Y)

From Figure 2.17 on page 87, we know that the text segment starts at address 40 0000_{hex} and the data segment at 1000 0000_{hex}. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is 40 0100_{hex}, and its data starts at 1000 0020_{hex}.

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have two types here:

1. The `jal`s are easy because they use pseudodirect addressing. The `jal` at address 40 0004_{hex} gets 40 0100_{hex} (the address of procedure B) in its address field, and the `jal` at 40 0104_{hex} gets 40 0000_{hex} (the address of procedure A) in its address field.
2. The load and store addresses are harder because they are relative to a base register. This example uses the global pointer as the base register. Figure 2.17 shows that `$gp` is initialized to 1000 8000_{hex}. To get the address 1000 0000_{hex} (the address of word X), we place 8000_{hex} in the address field of `lw` at address 40 0000_{hex}. Chapter 3 explains 16-bit two's complement computer arithmetic, which is why 8000_{hex} in the address field yields 1000 0000_{hex} as the address. Similarly, we place 8020_{hex} in the address field of `sw` at address 40 0100_{hex} to get the address 1000 0020_{hex} (the address of word Y).

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. It follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the machine registers and sets the stack pointer to the first free location.
6. Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

loader A systems program that places an object program in main memory so that it is ready to execute.

Sections A.3 and A.4 in  [Appendix A](#) describe linkers and **loaders** in more detail.

Dynamically Linked Libraries

The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

- The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
- It loads the whole library even if all of the library is not used when the program is run. The library can be large relative to the program; for example, the standard C library is 2.5 MB.

These disadvantages lead to dynamically linked libraries (DLLs), where the library routines are not linked and loaded until the program is run. Both the program and library routines keep extra information on the location of nonlocal procedures and their names. In the initial version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called versus those that are called during the running of the program. This observation led to the lazy procedure linkage version of DLLs, where each routine is linked only *after* it is called.

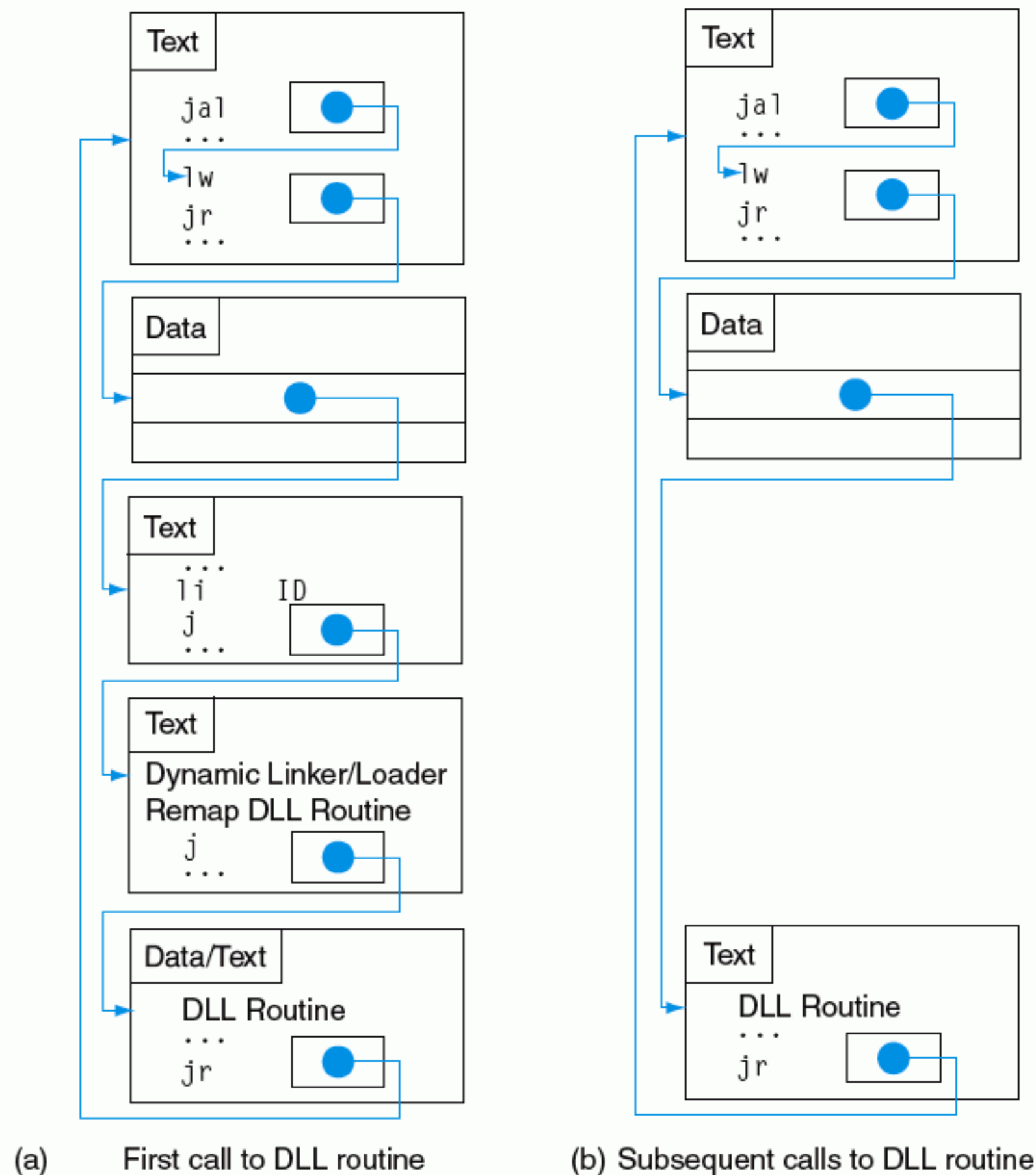


FIGURE 2.29 Dynamically linked library via lazy procedure linkage. (a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 7, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

Like many instances in our field, this trick relies on a level of indirection. Figure 2.29 shows the technique. It starts with the nonlocal routines calling a set of dummy routines at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect jump.

The first time the library routine is called, the program calls the dummy entry and follows the indirect jump. It points to code that puts a number in a register to identify the desired library routine and then jumps to the dynamic linker-loader.

The linker-loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine. It then jumps to it. When the routine completes, it returns to the original calling site. Thereafter, it jumps indirectly to the routine without the extra hops.

In summary, DLLs require extra space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect jump thereafter. Note that the return from the library pays no extra overhead. Microsoft’s Windows relies extensively on lazy dynamically linked libraries, and it is also the normal way of executing programs on UNIX systems today.

Starting a Java Program

The discussion above captures the traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a specific implementation of that architecture. Indeed, it is possible to execute Java programs just like C. Java was invented with a different set of goals, however. One was to quickly run safely on any computer, even if it might slow execution time.

Figure 2.30 shows the typical translation and execution steps for Java. Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the **Java bytecode** instruction set. This instruction set is designed to be close to the Java language so that this com-

Java bytecode Instruction from an instruction set designed to interpret Java programs.

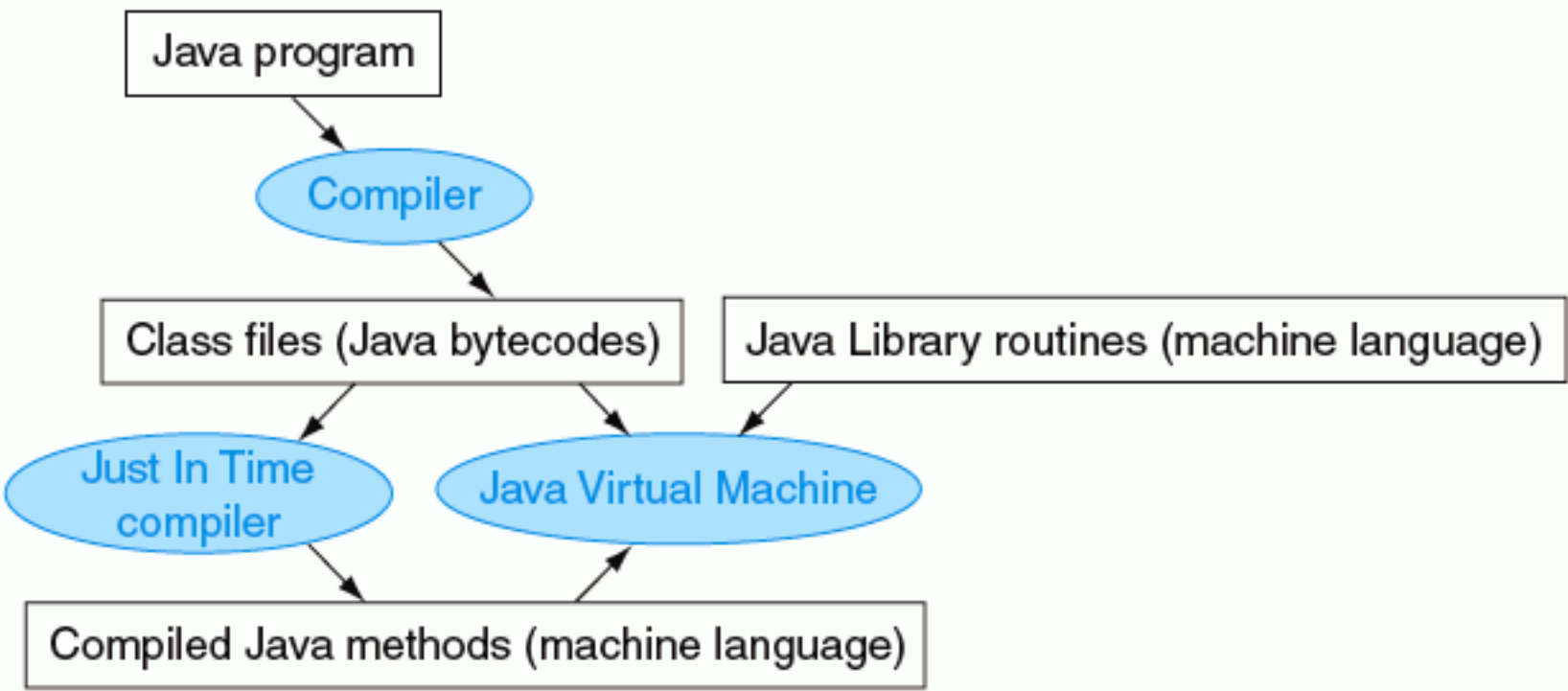


FIGURE 2.30 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the Java Virtual Machine (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the Just In Time (JIT) compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

pilation step is trivial. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

A software interpreter, called a **Java Virtual Machine** (JVM), can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the MIPS simulator used with this book is an interpreter. There is no need for a separate assembly step since either the translation is so simple that the compiler fills in the addresses or JVM finds them at runtime.

The upside of interpretation is portability. The availability of software Java virtual machines meant that most could write and run Java programs shortly after Java was announced. Today Java virtual machines are found in millions of devices, in everything from cell phones to Internet browsers.

The downside of interpretation is low performance. The incredible advances in performance of the 1980s and 1990s made interpretation viable for many important applications, but the factor of 10 slowdown when compared to traditionally compiled C programs made Java unattractive for some applications.

To preserve portability and improve execution speed, the next phase of Java development was compilers that translated *while* the program was running. Such **Just In Time compilers (JIT)** typically profile the running program to find where the “hot” methods are, and then compile them into the native instruction set on which the virtual machine is running. The compiled portion is saved for the next time the program is run, so that it can run faster each time it is run. This balance of interpretation and compilation evolves over time, so that frequently run Java programs suffer little of the overhead of interpretation.

As computers get faster so that compilers can do more, and as researchers invent better ways to compile Java on the fly, the performance gap between Java and C or C++ is closing. Section 2.14 goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

Which of the advantages of an interpreter over a translator do you think was most important for the designers of Java?

1. Ease of writing an interpreter
2. Better error messages
3. Smaller object code
4. Machine independence

Java Virtual Machine (JVM) The program that interprets Java bytecodes.

Just In Time Compiler (JIT) The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

Check Yourself

2.11 How Compilers Optimize

Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. The purpose of this section is to give a brief overview of optimizations a compiler uses to achieve performance. The following section introduces the internal anatomy of a compiler. To start, Figure 2.31 shows the structure of recent compilers, and we describe the optimizations in the order of the passes of that structure.

High-Level Optimizations

High-level optimizations are transformations that are done at something close to the source level.

The most common high-level transformation is probably *procedure inlining*, which replaces a call to a function by the body of the function, substituting the caller’s arguments for the procedure’s parameters. Other high-level optimizations

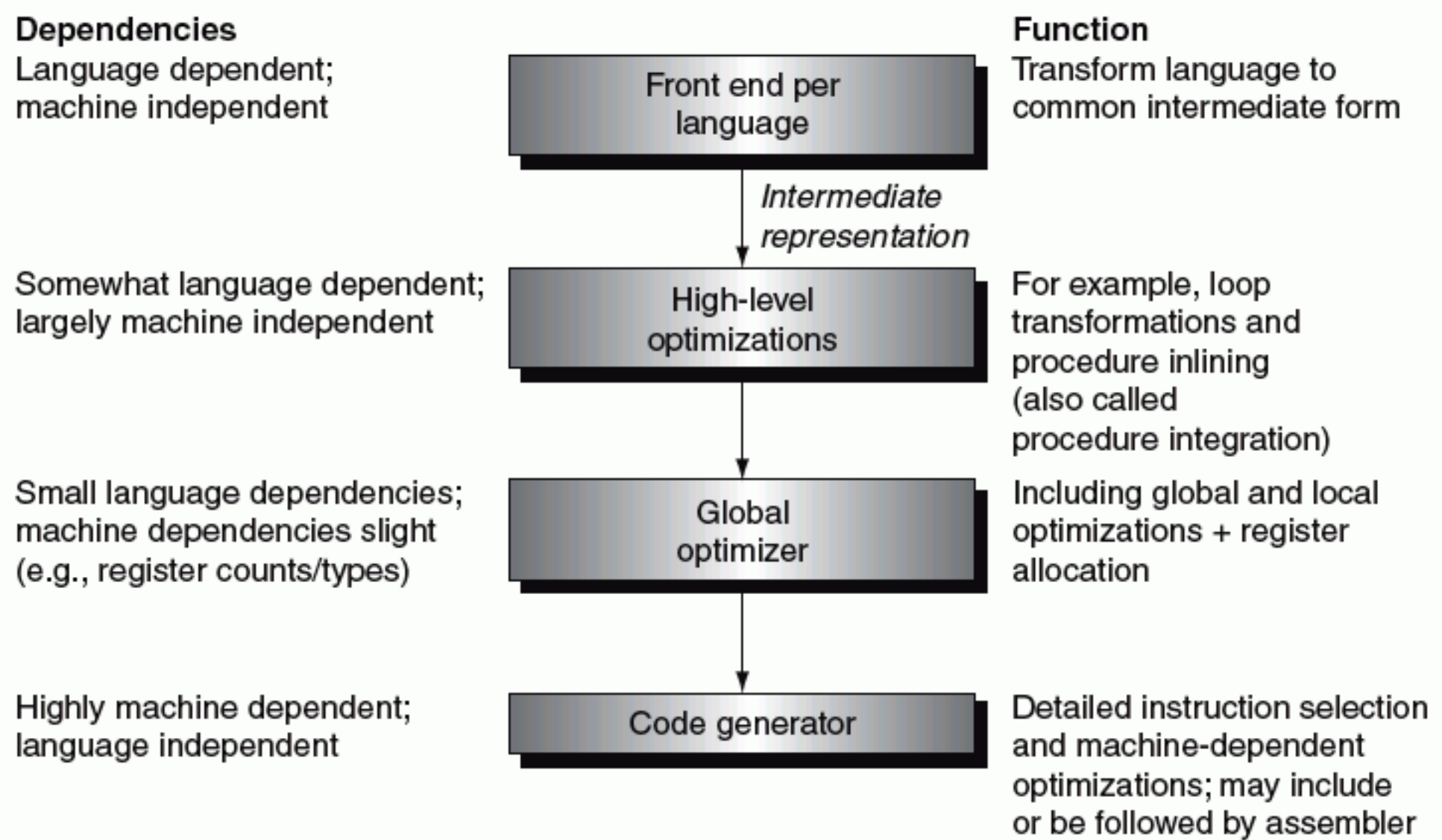


FIGURE 2.31 The structure of a modern optimizing compiler consists of a numbers of passes or phases. Logically each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle a procedure at a time, essentially interleaving with another pass.

involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively. For example, in loops that execute many iterations, such as those traditionally controlled by a *for* statement, the optimization of **loop unrolling** is often useful. Loop unrolling involves taking a loop and replicating the body multiple times and executing the transformed loop fewer times. Loop unrolling reduces the loop overhead and provides opportunities for many other optimizations. Other types of high-level transformations include sophisticated loop transformations such as interchanging nested loops and blocking loops to obtain better memory behavior; see Chapter 7 for examples.

loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

Local and Global Optimizations

Within the pass dedicated to local and global optimization, three classes of optimizations are performed:

1. *Local optimization* works within a single basic block. A local optimization pass is often run as a precursor and successor to global optimization to “clean up” the code before and after global optimization.
2. *Global optimization* works across multiple basic blocks; we will see an example of this shortly.
3. *Global register allocation* allocates variables to registers for regions of the code. Register allocation is crucial to getting good performance in modern processors.

Several optimizations are performed both locally as well as globally, including common subexpression elimination, constant propagation, copy propagation, dead store elimination, and strength reduction. Let’s look at some simple examples of these optimizations.

Common subexpression elimination finds multiple instances of the same expression and replaces the second one by a reference to the first. Consider, for example, a code segment to add 4 to an array element:

$$x[i] = x[i] + 4$$

The address calculation for $x[i]$ occurs twice and is identical since neither the starting address of x nor the value of i changes. Thus, the calculation can be reused. Let’s look at the intermediate code for this fragment, since it allows several other optimizations to be performed. Here is the unoptimized intermediate code on the left, and on the right is the code with common subexpression elimination replacing the second address calculation with the first. Note that the register allocation has not yet occurred, so the compiler is using virtual register numbers like R100 here.

<code># x[i] + 4</code>	<code># x[i] + 4</code>
<code>li R100,x</code>	<code>li R100,x</code>
<code>lw R101,i</code>	<code>lw R101,i</code>
<code>mult R102,R101,4</code>	<code>mult R102,R101,4</code>
<code>add R103,R100,R102</code>	<code>add R103,R100,R102</code>
<code>lw R104,0(R103)</code>	<code>lw R104,0(R103)</code>
<code># value of x[i] is in R104</code>	<code># value of x[i] is in R104</code>
<code>add R105,R104,4</code>	<code>add R105,R104,4</code>
<code># x[i] =</code>	<code># x[i] =</code>
<code>li R106,x</code>	<code>sw R105,0(R103)</code>
<code>lw R107,i</code>	
<code>mult R108,R107,4</code>	
<code>add R109,R106,R107</code>	
<code>sw R105,0(R109)</code>	

If the same optimization was possible across two basic blocks, it would then be an instance of *global common subexpression elimination*.

Let's consider some of the other optimizations:

- *Strength reduction* replaces complex operations by simpler ones and can be applied to this code segment, replacing the `mult` by a shift left.
- *Constant propagation* and its sibling *constant folding* find constants in code and propagates them, collapsing constant values whenever possible.
- *Copy propagation* propagates values that are simple copies, eliminating the need to reload values and possibly enabling other optimizations such as common subexpression elimination.
- *Dead store elimination* finds stores to values that are not used again and eliminates the store; its “cousin” is *dead code elimination*, which finds unused code—code that cannot affect the final result of the program—and eliminates it. With the heavy use of macros, templates, and the similar techniques designed to reuse code in high-level languages, dead code occurs surprisingly often.

Programmers concerned about performance of critical loops, especially in real-time or embedded applications, often find themselves staring at the assembly language produced by a compiler and wondering why the compiler failed to perform some global optimization or to allocate a variable to a register throughout a loop. The answer often lies in the dictate that the compiler be conservative. The opportunity for improving the code may seem obvious to the programmer, but then the programmer often has knowledge that the compiler does not have, such as the absence of aliasing between two pointers or the absence of side effects by a function call. The compiler may indeed be able to perform the transformation with a little help, which could eliminate the worst-case behavior that it must assume. This insight also illustrates an important observation: programmers who use pointers to try to improve performance in accessing variables, especially pointers to values on the stack that also have names as variables or as elements of arrays, are likely to disable many compiler optimizations. The end result is that the lower-level pointer code may run no better, or perhaps even worse, than the higher-level code optimized by the compiler.

Understanding Program Performance

Compilers must be *conservative*. The first task of a compiler is to produce correct code; its second task is usually to produce fast code although other factors such as code size may sometimes be important as well. Code that is fast but incorrect—for any possible combination of inputs—is simply wrong. Thus, when we say a compiler is “conservative,” we mean that it performs an optimization only if it knows with 100% certainty that, no matter what the inputs, the code will perform as the user wrote it. Since most compilers translate and optimize one function or procedure at a time, most compilers, especially at lower optimization levels, assume the worst about function calls and about their own parameters.

Global Code Optimizations

Many global code optimizations have the same aims as those used in the local case, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead code elimination.

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops. *Code motion* finds code that is loop invariant: a particular piece of code computes the same value on every loop iteration and, hence, may be computed once outside the loop. *Induction variable elimination* is a combination of

transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses. Rather than examine induction variable elimination in depth, we point the reader to Section 2.15, which compares the use of array indexing and pointers; for most loops, the transformation from the more obvious array code to the pointer code can be performed by a modern optimizing compiler.

Optimization Summary

Figure 2.32 gives examples of typical optimizations, and the last column indicates where the optimization is performed in the gcc compiler. It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator, and some optimizations are done multiple times, especially local optimizations, which may be performed before and after global optimization as well as during code generation.

Optimization name	Explanation	gcc level
High level	At or near the source level; processor independent	
Procedure integration	Replace procedure call by procedure body	O3
Local	Within straight-line code	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
Global	Across a branch	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., A = X) with X	O2
Code motion	Remove code from a loop that computes same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	Depends on processor knowledge	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

FIGURE 2.32 Major types of optimizations and examples in each class. The third column shows when these occur at different levels of optimization in gcc. The Gnu organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

Today essentially all programming for desktop and server applications is done in high-level languages, as is most programming for embedded applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. With Moore's law comes the temptation of adding sophisticated operations in an instruction set. The challenge is that they may not exactly match what the compiler needs to produce or be so general that they aren't fast. For example, consider special loop instructions found in some computers. Suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. The loop instruction may be a mismatch. When faced with such objections, the instruction set designer might then generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be slower than a sequence of simple operations.

Hardware Software Interface



How Compilers Work: An Introduction

The purpose of this section is to give a brief overview of the compiler function, which will help the reader understand both how the compiler translates a high-level language program into machine instructions. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course; our introduction will necessarily only touch on the basics. The rest of this section is on the CD.

2.13

A C Sort Example to Put It All Together

One danger of showing assembly language code in snippets is that you will have no idea what a full assembly language program looks like. In this section, we derive the MIPS code from two procedures written in C: one to swap array elements and one to sort them.

The Procedure `swap`

Let's start with the code for the procedure `swap` in Figure 2.33. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

This section describes the `swap` procedure in these three pieces, concluding by putting all the pieces together.

Register Allocation for `swap`

As mentioned on page 79, the MIPS convention on parameter passing is to use registers `$a0`, `$a1`, `$a2`, and `$a3`. Since `swap` has just two parameters, `v` and `k`, they will be found in registers `$a0` and `$a1`. The only other variable is `temp`, which we associate with register `$t0` since `swap` is a leaf procedure (see page 83). This register allocation corresponds to the variable declarations in the first part of the `swap` procedure in Figure 2.33.

Code for the Body of the Procedure `swap`

The remaining lines of C code in `swap` are

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Recall that the memory address for MIPS refers to the *byte* address, and so words are really 4 bytes apart. Hence we need to multiply the index `k` by 4 before

```
void swap(int v[], int k)  
{  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

FIGURE 2.33 A C procedure that swaps two locations in memory. The next subsection uses this procedure in a sorting example.

adding it to the address. *Forgetting that sequential word addresses differ by 4 instead of by 1 is a common mistake in assembly language programming.* Hence the first step is to get the address of $v[k]$ by multiplying k by 4:

```
sll    $t1, $a1, 2    # reg $t1 = k * 4
add    $t1, $a0, $t1  # reg $t1 = v + (k * 4)
                        # reg $t1 has the address of v[k]
```

Now we load $v[k]$ using $\$t1$, and then $v[k+1]$ by adding 4 to $\$t1$:

```
lw     $t0, 0($t1)    # reg $t0 (temp) = v[k]
lw     $t2, 4($t1)    # reg $t2 = v[k + 1]
                        # refers to next element of v
```

Next we store $\$t0$ and $\$t2$ to the swapped addresses:

```
sw     $t2, 0($t1)    # v[k] = reg $t2
sw     $t0, 4($t1)    # v[k+1] = reg $t0 (temp)
```

Now we have allocated registers and written the code to perform the operations of the procedure. What is missing is the code for preserving the saved registers used within swap. Since we are not using saved registers in this leaf procedure, there is nothing to preserve.

The Full swap Procedure

We are now ready for the whole routine, which includes the procedure label and the return jump. To make it easier to follow, we identify in Figure 2.34 each block of code with its purpose in the procedure.

The Procedure sort

To ensure that you appreciate the rigor of programming in assembly language, we'll try a second, longer example. In this case, we'll build a routine that calls the swap procedure. This program sorts an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts. Figure 2.35 shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

Register Allocation for sort

The two parameters of the procedure `sort`, v and n , are in the parameter registers $\$a0$ and $\$a1$, and we assign register $\$s0$ to i and register $\$s1$ to j .

Procedure body		
swap:	sll \$t1, \$a1, 2	# reg \$t1 = k * 4
	add \$t1, \$a0, \$t1	# reg \$t1 = v + (k * 4)
		# reg \$t1 has the address of v[k]
	lw \$t0, 0(\$t1)	# reg \$t0 (temp) = v[k]
	lw \$t2, 4(\$t1)	# reg \$t2 = v[k + 1]
		# refers to next element of v
	sw \$t2, 0(\$t1)	# v[k] = reg \$t2
	sw \$t0, 4(\$t1)	# v[k+1] = reg \$t0 (temp)
Procedure return		
	jr \$ra	# return to calling routine

FIGURE 2.34 MIPS assembly code of the procedure swap in Figure 2.33.

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

FIGURE 2.35 A C procedure that performs a sort on the array v.

Code for the Body of the Procedure sort

The procedure body consists of two nested *for* loops and a call to swap that includes parameters. Let’s unwrap the code from the outside to the middle.

The first translation step is the first *for* loop:

```
for (i = 0; i < n; i += 1) {
```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize *i* to 0, the first part of the *for* statement:

```
move    $s0, $zero    # i = 0
```

(Remember that move is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer; see page 107.) It also takes just one instruction to increment *i*, the last part of the *for* statement:

```
addi    $s0, $s0, 1    # i += 1
```

The loop should be exited if $i < n$ is *not* true or, said another way, should be exited if $i \geq n$. The set on less than instruction sets register $\$t0$ to 1 if $\$s0 < \$a1$ and 0 otherwise. Since we want to test if $\$s0 \geq \$a1$, we branch if register $\$t0$ is 0. This test takes two instructions:

```
for1tst:slt $t0, $s0, $a1    # reg $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq $t0, $zero, exit1 # go to exit1 if $s0 ≥ $a1 (i≥n)
```

The bottom of the loop just jumps back to the loop test:

```
        j for1tst           # jump to test of outer loop
exit1:
```

The skeleton code of the first *for* loop is then

```
        move $s0, $zero     # i = 0
for1tst:slt $t0, $s0, $a1    # reg $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq $t0, $zero, exit1 # go to exit1 if $s0 ≥ $a1 (i≥n)
        ...
        (body of first for loop)
        ...
        addi $s0, $s0, 1     # i += 1
        j for1tst           # jump to test of outer loop
exit1:
```

Voila! Exercise 2.14 explores writing faster code for similar loops.

The second *for* loop looks like this in C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

The initialization portion of this loop is again one instruction:

```
addi    $s1, $s0, -1 # j = i - 1
```

The decrement of j at the end of the loop is also one instruction:

```
addi    $s1, $s1, -1 # j -= 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails ($j < 0$):

```
for2tst:slti $t0, $s1, 0    # reg $t0 = 1 if $s1 < 0 (j < 0)
        bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
```

This branch will skip over the second condition test. If it doesn't skip, $j \geq 0$.

The second test exits if $v[j] > v[j + 1]$ is *not* true, or exits if $v[j] \leq v[j + 1]$. First we create the address by multiplying j by 4 (since we need a byte address) and add it to the base address of v :


```

sll    $t1, $s1, 2    # reg $t1 = j * 4
add    $t2, $a0, $t1  # reg $t2 = v + (j * 4)

```

Now we load $v[j]$:

```
lw     $t3, 0($t2)    # reg $t3 = v[j]
```

Since we know that the second element is just the following word, we add 4 to the address in register $\$t2$ to get $v[j + 1]$:

```
lw     $t4, 4($t2)    # reg $t4 = v[j + 1]
```

The test of $v[j] \leq v[j + 1]$ is the same as $v[j + 1] \geq v[j]$, so the two instructions of the exit test are

```

slt    $t0, $t4, $t3    # reg $t0 = 0 if $t4 ≥ $t3
beq    $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3

```

The bottom of the loop jumps back to the inner loop test:

```
j      for2tst    # jump to test of inner loop
```

Combining the pieces together, the skeleton of the second *for* loop looks like this:

```

        addi $s1, $s0, -1    # j = i - 1
for2tst: slti $t0, $s1, 0    # reg $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2    # reg $t1 = j * 4
        add  $t2, $a0, $t1  # reg $t2 = v + (j * 4)
        lw   $t3, 0($t2)    # reg $t3 = v[j]
        lw   $t4, 4($t2)    # reg $t4 = v[j + 1]
        slt  $t0, $t4, $t3  # reg $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
        ...
        (body of second for loop)
        ...
        addi $s1, $s1, -1    # j -= 1
        j    for2tst        # jump to test of inner loop
exit2:

```

The Procedure Call in `sort`

The next step is the body of the second *for* loop:

```
swap(v, j);
```

Calling swap is easy enough:

```
jal    swap
```

Passing Parameters in sort

The problem comes when we want to pass parameters because the sort procedure needs the values in registers \$a0 and \$a1, yet the swap procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for sort into other registers earlier in the procedure, making registers \$a0 and \$a1 available for the call of swap. (This copy is faster than saving and restoring on the stack.) We first copy \$a0 and \$a1 into \$s2 and \$s3 during the procedure:

```
move    $s2, $a0    # copy parameter $a0 into $s2
move    $s3, $a1    # copy parameter $a1 into $s3
```

Then we pass the parameters to swap with these two instructions:

```
move    $a0, $s2    # first swap parameter is v
move    $a1, $s1    # second swap parameter is j
```

Preserving Registers in sort

The only remaining code is the saving and restoring of registers. Clearly we must save the return address in register \$ra, since sort is a procedure and is called itself. The sort procedure also uses the saved registers \$s0, \$s1, \$s2, and \$s3, so they must be saved. The prologue of the sort procedure is then

```
addi    $sp, $sp, -20    # make room on stack for 5 regs
sw      $ra, 16($sp)     # save $ra on stack
sw      $s3, 12($sp)     # save $s3 on stack
sw      $s2, 8($sp)      # save $s2 on stack
sw      $s1, 4($sp)      # save $s1 on stack
sw      $s0, 0($sp)      # save $s0 on stack
```

The tail of the procedure simply reverses all these instructions, then adds a jr to return.

The Full Procedure sort

Now we put all the pieces together in Figure 2.36, being careful to replace references to registers \$a0 and \$a1 in the *for* loops with references to registers \$s2 and \$s3. Once again to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, 9 lines of the sort procedure in C became 35 lines in the MIPS assembly language.

Saving registers			
sort:	addi	\$sp,\$sp,-20	# make room on stack for 5 registers
	sw	\$ra,16(\$sp)	# save \$ra on stack
	sw	\$s3,12(\$sp)	# save \$s3 on stack
	sw	\$s2,8(\$sp)	# save \$s2 on stack
	sw	\$s1,4(\$sp)	# save \$s1 on stack
	sw	\$s0,0(\$sp)	# save \$s0 on stack
Procedure body			
Move parameters	move	\$s2,\$a0	# copy parameter \$a0 into \$s2 (save \$a0)
	move	\$s3,\$a1	# copy parameter \$a1 into \$s3 (save \$a1)
Outer loop	move	\$s0,\$zero	# i = 0
	for1tst:slt	\$t0,\$s0,\$s3	# reg \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)
	beq	\$t0,\$zero,exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)
Inner loop	addi	\$s1,\$s0,-1	# j = i - 1
	for2tst:slti	\$t0,\$s1,0	# reg \$t0 = 1 if \$s1 < 0 (j < 0)
	bne	\$t0,\$zero,exit2	# go to exit2 if \$s1 < 0 (j < 0)
	sll	\$t1,\$s1,2	# reg \$t1 = j * 4
	add	\$t2,\$s2,\$t1	# reg \$t2 = v + (j * 4)
	lw	\$t3,0(\$t2)	# reg \$t3 = v[j]
	lw	\$t4,4(\$t2)	# reg \$t4 = v[j + 1]
	slt	\$t0,\$t4,\$t3	# reg \$t0 = 0 if \$t4 ≥ \$t3
	beq	\$t0,\$zero,exit2	# go to exit2 if \$t4 ≥ \$t3
Pass parameters and call	move	\$a0,\$s2	# 1st parameter of swap is v (old \$a0)
	move	\$a1,\$s1	# 2nd parameter of swap is j
	jal	swap	# swap code shown in Figure 2.34
Inner loop	addi	\$s1,\$s1,-1	# j -= 1
	j	for2tst	# jump to test of inner loop
Outer loop	exit2: addi	\$s0,\$s0,1	# i += 1
	j	for1tst	# jump to test of outer loop
Restoring registers			
exit1:	lw	\$s0,0(\$sp)	# restore \$s0 from stack
	lw	\$s1,4(\$sp)	# restore \$s1 from stack
	lw	\$s2,8(\$sp)	# restore \$s2 from stack
	lw	\$s3,12(\$sp)	# restore \$s3 from stack
	lw	\$ra,16(\$sp)	# restore \$ra from stack
	addi	\$sp,\$sp,20	# restore stack pointer
Procedure return			
	jr	\$ra	# return to calling routine

FIGURE 2.36 MIPS assembly version of procedure sort in Figure 2.35 on page 124.

Elaboration: One optimization that works with this example is *procedure inlining*, mentioned in Section 2.11. Instead of passing arguments in parameters and invoking the code with a `jal` instruction, the compiler would copy the code from the body of the swap procedure where the call to swap appears in the code. Inlining would avoid four

instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger if the inlined procedure is called from several locations. Such a code expansion might turn into *lower* performance if it increased the cache miss rate; see Chapter 7.

The MIPS compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement `$sp` by 16 to make room for all four argument registers (16 bytes). One reason is that C provides a `vararg` option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare `vararg`, it copies the four argument registers onto the stack into the four reserved locations.

Figure 2.37 shows the impact of compiler optimization on sort program performance, compile time, clock cycles, instruction count, and CPI. Note that unoptimized code has the best CPI and O1 optimization has the lowest instruction count, but O3 is the fastest, reminding us that time is the only accurate measure of program performance.

Figure 2.38 compares the impact of programming languages, compilation versus interpretation, and algorithms on performance of sorts. The fourth column shows that the unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort. Using the Just In Time Java compiler makes Java 2.1 times *faster* than the unoptimized C and within a factor of 1.13 of the highest optimized C code. (The next section gives more details on interpretation versus compilation of Java and the Java and MIPS code for Bubble Sort.) The ratios aren't as close for Quicksort in column 5, presumably because it is harder to amortize the cost of runtime compilation over the shorter execution time. The last column demonstrates the impact of a better algorithm, offering three orders of magnitude performance increase when sorting 100,000 items. Even comparing interpreted Java in column 5 to the C compiler at highest optimization in column 4, Quicksort beats Bubble Sort by a factor of 50 (0.05×2468 or 123 versus 2.41).

Understanding Program Performance

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
none	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.37 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM memory. It used Linux version 2.4.20.

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	compiler	none	1.00	1.00	2468
	compiler	01	2.37	1.50	1562
	compiler	02	2.38	1.50	1555
	compiler	03	2.41	1.91	1955
Java	interpreter	—	0.12	0.05	1050
	Just In Time compiler	—	2.13	0.29	338

FIGURE 2.38 Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version. The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as Figure 2.37. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.



Implementing an Object-Oriented Language

object-oriented language A programming language that is oriented around objects rather than actions, or data versus logic.

This section is for readers interested in seeing how an **object-oriented language** like Java executes on a MIPS architecture. It shows the Java bytecodes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort. The rest of this section is on the CD.



Arrays versus Pointers

A challenging topic for any new programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insights about pointers. This section shows C and MIPS assembly versions of two procedures to clear a sequence of words in memory: one using array indices and one using pointers. Figure 2.39 shows the two C procedures.

The purpose of this section is to show how pointers map into MIPS instructions, and not to endorse a dated programming style. We'll see the impact of modern compiler optimization on these two procedures at the end of the section.

Array Version of Clear

Let's start with the array version, `clear1`, focusing on the body of the loop and ignoring the procedure linkage code. We assume that the two parameters `array` and `size` are found in the registers `$a0` and `$a1`, and that `i` is allocated to register `$t0`.

```

clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}

```

FIGURE 2.39 Two C procedures for setting an array to all zeros. `clear1` uses indices, while `clear2` uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by `&`, and referring to the object pointed to by a pointer is indicated by `*`. The declarations declare that `array` and `p` are pointers to integers. The first part of the *for* loop in `clear2` assigns the address of the first element of `array` to the pointer `p`. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of `array`. Incrementing a pointer by one, in the last part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since `p` is a pointer to integers, the compiler will generate MIPS instructions to increment `p` by four, the number of bytes in a MIPS integer. The assignment in the loop places 0 in the object pointed to by `p`.

The initialization of `i`, the first part of the *for* loop, is straightforward:

```
move    $t0,$zero    # i = 0 (register $t0 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 4 to get the byte address:

```
loop1: sll    $t1,$t0,2    # $t1 = i * 4
```

Since the starting address of the array is in a register, we must add it to the index to get the address of `array[i]` using an add instruction:

```
add     $t2,$a0,$t1    # $t2 = address of array[i]
```

(This example is an ideal situation for indexed addressing; see [In More Depth](#) in Section 2.20 on page 147.) Finally, we can store 0 in that address:

```
sw      $zero, 0($t2)  # array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment `i`:

```
addi    $t0,$t0,1      # i = i + 1
```


The loop test checks if *i* is less than *size*:

```
slt    $t3,$t0,$a1    # $t3 = (i < size)
bne    $t3,$zero,loop1 # if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the MIPS code for clearing an array using indices:

```
move    $t0,$zero      # i = 0
loop1: sll    $t1,$t0,2   # $t1 = i * 4
add     $t2,$a0,$t1     # $t2 = address of array[i]
sw      $zero, 0($t2)    # array[i] = 0
addi    $t0,$t0,1       # i = i + 1
slt     $t3,$t0,$a1     # $t3 = (i < size)
bne     $t3,$zero,loop1 # if (i < size) go to loop1
```

(This code works as long as *size* is greater than 0.)

Pointer Version of Clear

The second procedure that uses pointers allocates the two parameters *array* and *size* to the registers *\$a0* and *\$a1* and allocates *p* to register *\$t0*. The code for the second procedure starts with assigning the pointer *p* to the address of the first element of the array:

```
move    $t0,$a0        # p = address of array[0]
```

The next code is the body of the *for* loop, which simply stores 0 into *p*:

```
loop2: sw    $zero,0($t0) # Memory[p] = 0
```

This instruction implements the body of the loop, so the next code is the iteration increment, which changes *p* to point to the next word:

```
addi    $t0,$t0,4       # p = p + 4
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since *p* is a pointer to integers, each of which use 4 bytes, the compiler increments *p* by 4.

The loop test is next. The first step is calculating the address of the last element of array. Start with multiplying *size* by 4 to get its byte address:

```
add     $t1,$a1,$a1     # $t1 = size * 2
add     $t1,$t1,$t1     # $t1 = size * 4
```

and then we add the product to the starting address of the array to get the address of the first word *after* the array:

```
add $t2,$a0,$t1      # $t2 = address of array[size]
```

The loop test is simply to see if *p* is less than the last element of array:

```
slt $t3,$t0,$t2      # $t3 = (p<&array[size])
bne $t3,$zero,loop2  # if (p<&array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
move $t0,$a0          # p = address of array[0]
loop2:sw $zero,0($t0)  # Memory[p] = 0
addi $t0,$t0,4        # p = p + 4
add $t1,$a1,$a1       # $t1 = size * 2
add $t1,$t1,$t1       # $t1 = size * 4
add $t2,$a0,$t1       # $t2 = address of array[size]
slt $t3,$t0,$t2       # $t3 = (p<&array[size])
bne $t3,$zero,loop2   # if (p<&array[size]) go to loop2
```

As in the first example, this code assumes *size* is greater than 0.

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
move $t0,$a0          # p = address of array[0]
sll $t1,$a1,2         # $t1 = size * 4
add $t2,$a0,$t1       # $t2 = address of array[size]
loop2:sw $zero,0($t0)  # Memory[p] = 0
addi $t0,$t0,4        # p = p + 4
slt $t3,$t0,$t2       # $t3 = (p<&array[size])
bne $t3,$zero,loop2   # if (p<&array[size]) go to loop2
```

Comparing the Two Versions of Clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

move \$t0,\$zero	# i = 0	move \$t0,\$a0	# p = & array[0]
loop1:sll \$t1,\$t0,2	# \$t1 = i * 4	sll \$t1,\$a1,2	# \$t1 = size * 4
add \$t2,\$a0,\$t1	# \$t2 = &array[i]	add \$t2,\$a0,\$t1	# \$t2 = &array[size]
sw \$zero, 0(\$t2)	# array[i] = 0	loop2:sw \$zero,0(\$t0)	# Memory[p] = 0
addi \$t0,\$t0,1	# i = i + 1	addi \$t0,\$t0,4	# p = p + 4
slt \$t3,\$t0,\$a1	# \$t3 = (i < size)	slt \$t3,\$t0,\$t2	# \$t3=(p<&array[size])
bne \$t3,\$zero,loop1	# if () go to loop1	bne \$t3,\$zero,loop2	# if () go to loop2

The version on the left must have the “multiply” and add inside the loop because `i` is incremented and each address must be recalculated from the new index; the memory pointer version on the right increments the pointer `p` directly. The pointer version reduces the instructions executed per iteration from 7 to 4. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops).

Elaboration: The C compiler would add a test to be sure that `size` is greater than 0. One way would be to add a jump just before the first instruction of the loop to the `slt` instruction.

Understanding Program Performance

People used to be taught to use pointers in C to get greater efficiency than available with arrays: “Use pointers, even if you can’t understand the code.” Modern optimizing compilers can produce just as good code for the array version. Most programmers today prefer that the compiler do the heavy lifting.

*Beauty is altogether in the
eye of the beholder.*

Margaret Wolfe Hungerford,
Molly Bawn, 1877

2.16

Real Stuff: IA-32 Instructions

Designers of instruction sets sometimes provide more powerful operations than those found in MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence (see Section 4.8).

The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions. Section 2.17 demonstrates the pitfalls of complexity.

The Intel IA-32

MIPS was the vision of a single small group in 1985; the pieces of this architecture fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the IA-32; it is the product of several independent groups who evolved the architecture over almost 20 years, adding new features to the original

instruction set as someone might add clothing to a packed bag. Here are important IA-32 milestones:

- **1978:** The Intel 8086 architecture was announced as an assembly-language-compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike MIPS, the registers have dedicated uses, and hence the 8086 is not considered a **general-purpose register** architecture.
- **1980:** The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see Section 2.19 and Section 3.9).
- **1982:** The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see Chapter 7), and by adding a few instructions to round out the instruction set and to manipulate the protection model.
- **1985:** The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 7). Like the 80286, the 80386 has a mode to execute 8086 programs without change.
- **1989–95:** The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (Chapter 9) and a conditional move instruction.
- **1997:** After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX (Multi Media Extensions). This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the tradition of single instruction, multiple data (SIMD) architectures (see Chapter 9). Pentium II did not introduce any new instructions.
- **1999:** Intel added another 70 instructions, labeled SSE (Streaming SIMD Extensions) as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single-precision floating-point data type. Hence four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE included cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

general-purpose register (GPR) A register that can be used for addresses or for data with virtually any instruction.

- **2001:** Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double-precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable more multimedia operations, it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers like those found in other computers. This change has boosted floating-point performance on the Pentium 4, the first microprocessor to include SSE2 instructions.
- **2003:** A company other than Intel enhanced the IA-32 architecture this time. AMD announced a set of architectural extensions to increase the address space from 32 to 64 bits. Similar to the transition from a 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and increases the number of 128-bit SSE registers to 16. The primary ISA change comes from adding a new mode called *long mode* that redefines the execution of all IA-32 instructions with 64-bit addresses and data. To address the larger number of registers, it adds a new prefix to instructions. Depending how you count, long mode also adds 4 to 10 new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to IA-32 (*legacy mode*) plus a mode that restricts user programs to IA-32 but allows operating systems to use AMD64 (*compatibility mode*). These modes allow a more graceful transition to 64-bit addressing than the HP/Intel IA-64 architecture.
- **2004:** Intel capitulates and embraces AMD64, relabeling it Extended Memory 64 Technology (EM64T). The major difference is that Intel added a 128-bit atomic compare and swap instruction, which probably should have been included in AMD64. At the same time, Intel announced another generation of media extensions. SSE3 adds 13 instructions to support complex arithmetic, graphics operations on arrays of structures, video encoding, floating point conversion, and thread synchronization (see Chapter 9). AMD will offer SSE3 in subsequent chips and it will almost certainly add the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.

This history illustrates the impact of the “golden handcuffs” of compatibility on the IA-32, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the IA-32, keep in mind that there are more instances of this architectural family on desktops than of any other architecture, increasing by 100 million per year. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write IA-32 programs; the goal instead is to give

you familiarity with the strengths and weaknesses of the world’s most popular desktop architecture.

Rather than show the entire 16-bit and 32-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386, as this portion of the architecture is what is used. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

IA-32 Registers and Data Addressing Modes

The registers of the 80386 shows the evolution of the instruction set (Figure 2.40). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits,



FIGURE 2.40 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

FIGURE 2.41 Instruction types for the arithmetic, logical, and data transfer instructions. The IA-32 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.40 (not EIP or EFLAGS).

prefixing an *E* to their name to indicate the 32-bit version. We'll refer to them generically as GPRs (general-purpose registers). The 80386 contains only eight GPRs. This means MIPS programs can use four times as many.

The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure 2.41. There are two important differences here. The IA-32 arithmetic and logical instructions must have one operand act as both a source and a destination; MIPS allows separate registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus virtually any instruction may have one operand in memory, unlike MIPS and PowerPC.

The seven data memory-addressing modes, described in detail below, offer two sizes of addresses within the instruction. These so-called *displacements* can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. Figure 2.42 shows the IA-32 addressing modes and which GPRs cannot be used with that mode, plus how you would get the same effect using MIPS instructions.

IA-32 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The 80386 adds 32-bit addresses and data (*double words*) in the IA-32. The data type distinctions apply to register operations as well as memory accesses. Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode and is either 16 bits or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80386 architects provide a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominates

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	lw \$s0,0(\$s1)
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	lw \$s0,100(\$s1) # ≤16-bit displacement
Base plus scaled index	The address is Base + (2 ^{Scale} × Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2 ^{Scale} × Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) # ≤16-bit displacement

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

most programs, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus to support a semaphore (see Chapter 9), or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The IA-32 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including test, integer, and decimal arithmetic operations
3. Control flow, including conditional branches, unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location. Figure 2.43 shows some typical IA-32 instructions and their functions.

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Conditional branches on the IA-32 are based on *condition codes* or *flags*. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. The argument for condition codes is that they occur as part of normal operations and are faster to test than it is to compare registers, as MIPS does for beq and bne. The argument against condition codes is that the compare to 0 extends the time of the operation, since it uses extra hardware after the operation, and that often the programmer must use compare instructions to test a value that is not the result of an operation. Moreover, PC-relative branch addresses must be specified in the number of bytes, since unlike MIPS, 80386 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the IA-32 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the fallacy on page 143).

Figure 2.44 lists some of the integer IA-32 instructions. Many of the instructions are available in both byte and word formats.

IA-32 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 80836 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 17 bytes.

Figure 2.45 shows the instruction format for several of the example instructions in Figure 2.43. The opcode byte usually contains a bit saying whether the operand is

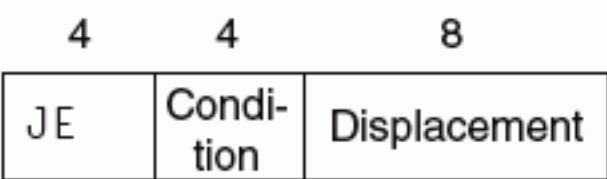
Instruction	Meaning
Control	Conditional and unconditional branches
JNZ, JZ	Jump if condition to EIP + 8-bit offset; JNE (for JNZ), JE (for JZ) are alternative names
JMP	Unconditional jump—8-bit or 16-bit offset
CALL	Subroutine call—16-bit offset; return address pushed onto stack
RET	Pops return address from stack and jumps to it
LOOP	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	Move data between registers or between register and memory
MOV	Move between two registers or between register and memory
PUSH, POP	Push source operand on stack; pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic, logical	Arithmetic and logical operations using the data registers and memory
ADD, SUB	Add source to destination; subtract source from destination; register-memory format
CMP	Compare source and destination; register-memory format
SHL, SHR, RCR	Shift left; shift logical right; rotate right with carry condition code as fill
CBW	Convert byte in 8 rightmost bits of EAX to 16-bit word in right of EAX
TEST	Logical AND of source and destination sets condition codes
INC, DEC	Increment destination, decrement destination
OR, XOR	Logical OR; exclusive OR; register-memory format
String	Move between string operands; length given by a repeat prefix
MOVS	Copies from string source to destination by incrementing ESI and EDI; may be repeated
LODS	Loads a byte, word, or double word of a string into the EAX register

FIGURE 2.44 Some typical operations on the IA-32. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

8 bits or 32 bits. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form “register = register op immediate.” Other instructions use a “postbyte” or extra opcode byte, labeled “mod, reg, r/m,” which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled “sc, index, base.”

Figure 2.46 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to fully understand which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes even the encoding of the instructions.

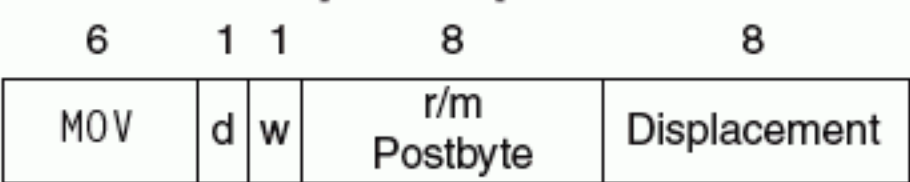
a. JE EIP + displacement



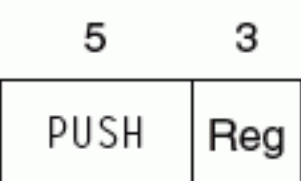
b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42

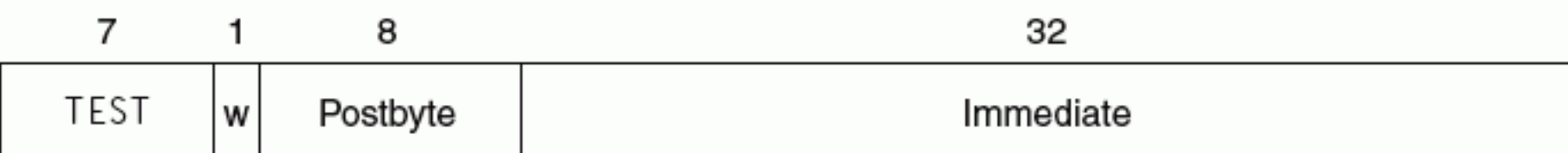


FIGURE 2.45 Typical IA-32 instruction formats. Figure 2.46 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or double word. The d field in MOV is used in instructions that may move to or from memory and shows the direction of the move. The ADD instruction requires 32 bits for the immediate field because in 32-bit mode the immediates are either 8 bits or 32 bits. The immediate field in the TEST is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 17 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

IA-32 Conclusion

Intel had a 16-bit microprocessor two years before its competitors’ more elegant architectures, such as the Motorola 68000, and this headstart led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the IA-32 is more difficult to build than machines like MIPS, but the much

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	same	same	same	same	same
1	CL	CX	ECX	1	addr=BX+DI	=ECX	addr as	addr as	addr as	addr as	as
2	DL	DX	EDX	2	addr=BP+SI	=EDX	mod=0	mod=0	mod=0	mod=0	reg
3	BL	BX	EBX	3	addr=BP+SI	=EBX	+ disp8	+ disp8	+ disp16	+ disp32	field
4	AH	SP	ESP	4	addr=SI	=(sib)	SI+disp8	(sib)+disp8	SI+disp8	(sib)+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

FIGURE 2.46 The encoding of the first address specifier of the IA-32, “mod, reg, r/m.” The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod ≠ 3, where (sib) means use the scaled index mode shown in Figure 2.42. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

larger market means Intel can afford more resources to help overcome the added complexity. What the IA-32 lacks in style is made up in quantity, making it beautiful from the right perspective.

The saving grace is that the most frequently used IA-32 architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

2.17 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel IA-32 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the following instruction until a counter counts down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to

memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times faster. A third version, which used the larger floating-point registers instead of the integer registers of the IA-32, copies at about 2.0 times faster than the complex instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naive instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to thoroughly understand the concepts in Chapters 6 and 7 (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is one situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore such hints because the compiler does a better job at allocation than the programmer.

Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are longer time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to future machines, it also makes the software easier to maintain and allows the program to run on more brands of computers.

Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in Figure 2.16, the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

2.18

Concluding Remarks*Less is more.*Robert Browning, *Andrea del Sarto*, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid environmental scientists, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. Four design principles guide the authors of instruction sets in making that delicate balance:


1. *Simplicity favors regularity.* Regularity motivates many features of the MIPS instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.
2. *Smaller is faster.* The desire for speed is the reason that MIPS has 32 registers rather than many more.
3. *Make the common case fast.* Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for constant operands.
4. *Good design demands good compromises.* One MIPS example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even “extends” the instruction set by creating symbolic instructions that aren’t in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. Figure 2.47 lists the MIPS instructions we have covered so far, both real and pseudoinstructions.

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.48 shows the popularity of each class of instructions for SPEC2000. The varying popularity of instructions plays an important role in the chapters on performance, datapath, control, and pipelining.

Each category of MIPS instructions is associated with constructs that appear in programming languages:

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
store byte	sb	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURE 2.47 The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right.  Appendix A (Section A.10, on page A-45) describes the full MIPS architecture. Figure 2.27 shows more details of the MIPS architecture revealed in this chapter.

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	operations in assignment statements	24%	48%
Data transfer	lw, sw, lb, sb, lui	references to data structures, such as arrays	36%	39%
Logical	and, or, nor, andi, ori, sll, srl	operations in assignment statements	18%	4%
Conditional branch	beq, bne, slt, slti	if statements and loops	18%	6%
Jump	j, jr, jal	procedure calls, returns, and case/switch statements	3%	0%

FIGURE 2.48 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for average of five SPEC2000 integer programs and five SPEC2000 floating point programs. Figure 3.26 shows the percentage of the individual MIPS instructions executed.

- The arithmetic instructions correspond to the operations found in assignment statements.
- Data transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- The conditional branches are used in *if* statements and in loops.
- The unconditional jumps are used in procedure calls and returns and for *case/switch* statements.

After we explain computer arithmetic in Chapter 3, we reveal more of the MIPS instruction set architecture.



Historical Perspective and Further Reading



This section surveys the history of instruction set architectures over time, and we give a short history of programming languages and compilers. ISAs include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of the IA-32. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes the key milestones and the pioneers who achieved them. The rest of this section is on the CD.

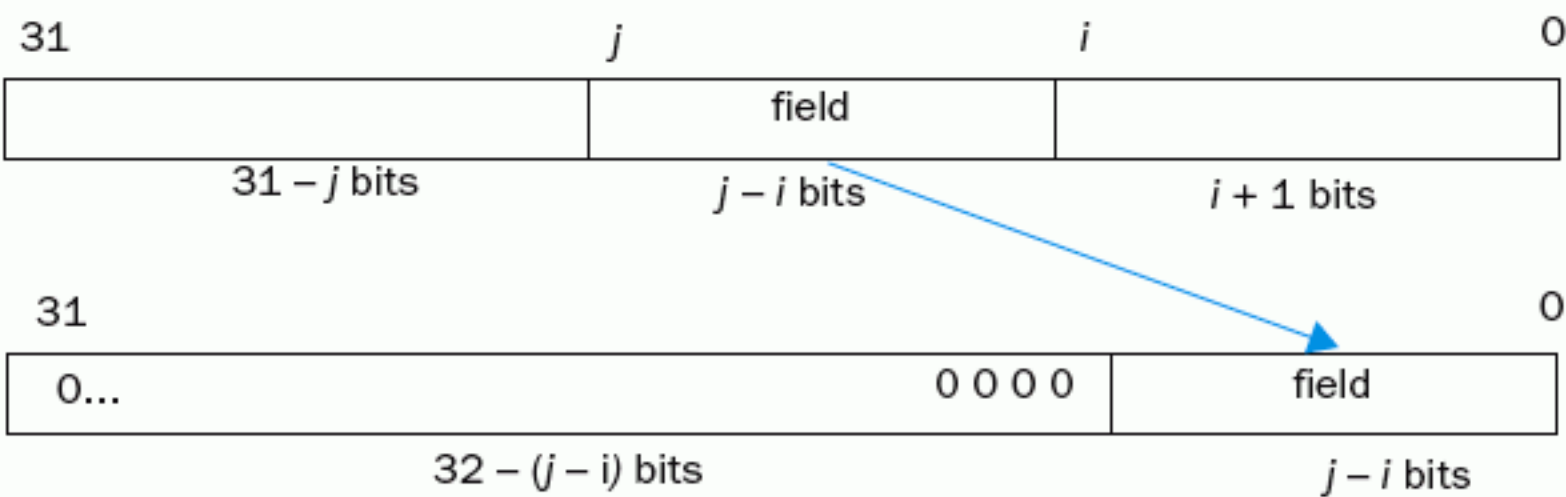
2.20







Exercises

🔗 [Appendix A](#) describes the MIPS simulator, which is helpful for these exercises. Although the simulator accepts pseudoinstructions, try not to use pseudoinstructions for any exercises that ask you to produce MIPS code. Your goal should be to learn the real MIPS instruction set, and if you are asked to count instructions, your count should reflect the actual instructions that will be executed and not the pseudoinstructions.

There are some cases where pseudoinstructions must be used (for example, the `l a` instruction when an actual value is not known at assembly time). In many cases, they are quite convenient and result in more readable code (for example, the `l i` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why.

- 2.1 [15] <§2.4>  For More Practice: Instruction Formats
- 2.2 [5] <§2.4> What binary number does this hexadecimal number represent: $7fff\ fffa_{hex}$? What decimal number does it represent?
- 2.3 [5] <§2.4> What hexadecimal number does this binary number represent: $1100\ 1010\ 1111\ 1110\ 1111\ 1010\ 1100\ 1110_{two}$?
- 2.4 [5] <§2.4> Why doesn't MIPS have a subtract immediate instruction?
- 2.5 [15] <§2.5>  For More Practice: MIPS Code and Logical Operations
- 2.6 [15] <§2.5> Some computers have explicit instructions to extract an arbitrary field from a 32-bit register and to place it in the least significant bits of a register. The figure below shows the desired operation:



- Find the shortest sequence of MIPS instructions that extracts a field for the constant values $i = 5$ and $j = 22$ from register $\$t3$ and places it in register $\$t0$. (Hint: It can be done in two instructions.)
- 2.7 [10] <§2.5>  For More Practice: Logical Operations in MIPS
- 2.8 [20] <§2.5>  In More Depth: Bit Fields in C
- 2.9 [20] <§2.5>  In More Depth: Bit Fields in C
- 2.10 [20] <§2.5>  In More Depth: Jump Tables
- 2.11 [20] <§2.5>  In More Depth: Jump Tables
- 2.12 [20] <§2.5>  In More Depth: Jump Tables

2.13 [10] <§2.6> Construct a control flow graph (like the one shown in Fig. 2.11) for the following section of C or Java code:

```
for (i=0; i<x; i=i+1)
    y = y + i;
```

2.14 [10] <§2.6>  [For More Practice](#): Writing Assembly Code

2.15 [25] <§2.7> Implement the following C code in MIPS, assuming that `set_array` is the first function called:

```
int i;
void set_array(int num) {
    int array[10];
    for (i=0; i<10; i++) {
        array[i] = compare(num, i);
    }
}
int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}
int sub (int a, int b) {
    return a-b;
}
```

Be sure to handle the stack and frame pointers appropriately. The variable `code` is allocated on the stack, and `i` corresponds to `$s0`. Draw the status of the stack before calling `set_array` and during each function call. Indicate the names of registers and variables stored on the stack and mark the location of `$sp` and `$fp`.

2.16 [30] <§2.7>  [In More Depth](#): Tail Recursion

2.17 [30] <§2.7>  [In More Depth](#): Tail Recursion

2.18 [20] <§2.7>  [In More Depth](#): Tail Recursion

2.19 [5] <§2.8> Iris and Julie are students in computer engineering who are learning about ASCII and Unicode character sets. Help them by spelling their names and your first name in both ASCII (using decimal notation) and Unicode (using hex notation and the Basic Latin character set).

2.20 [10] <§2.8> Compute the decimal byte values that form the null-terminated ASCII representation of the following string:

A byte is 8 bits

2.21 [30] <§§2.7, 2.8>  [For More Practice](#): MIPS Coding and ASCII Strings

2.22 [20] <§§2.7, 2.8>  [For More Practice](#): MIPS Coding and ASCII Strings

2.23 [20] <§§2.7, 2.8> {Ex. 2.22}  [For More Practice](#): MIPS Coding and ASCII Strings

2.24 [30] <§§2.7, 2.8>  [For More Practice](#): MIPS Coding and ASCII Strings

2.25 <§2.8>  [For More Practice](#): Comparing C/Java to MIPS

2.26 <§2.8>  [For More Practice](#): Translating MIPS to C

2.27 <§2.8>  [For More Practice](#): Understanding MIPS Code

2.28 <§2.8>  [For More Practice](#): Understanding MIPS Code

2.29 [5] <§§2.3, 2.6, 2.9> Add comments to the following MIPS code and describe in one sentence what it computes. Assume that \$a0 and \$a1 are used for the input and both initially contain the integers *a* and *b*, respectively. Assume that \$v0 is used for the output.

```

                                add    $t0, $zero, $zero
loop:                          beq     $a1, $zero, finish
                                add     $t0, $t0, $a0
                                sub     $a1, $a1, 1
                                j       loop
finish:                        addi    $t0, $t0, 100
                                add     $v0, $t0, $zero
```

2.30 [12] <§§2.3, 2.6, 2.9> The following code fragment processes two arrays and produces an important value in register \$v0. Assume that each array consists of 2500 words indexed 0 through 2499, that the base addresses of the arrays are stored in \$a0 and \$a1 respectively, and their sizes (2500) are stored in \$a2 and \$a3, respectively. Add comments to the code and describe in one sentence what this code does. Specifically, what will be returned in \$v0?

```

                                sll     $a2, $a2, 2
                                sll     $a3, $a3, 2
                                add     $v0, $zero, $zero
                                add     $t0, $zero, $zero
outer:                          add     $t4, $a0, $t0
```

```

        lw      $t4, 0($t4)
        add     $t1, $zero, $zero
inner:   add     $t3, $a1, $t1
        lw      $t3, 0($t3)
        bne     $t3, $t4, skip
        addi    $v0, $v0, 1
skip:    addi    $t1, $t1, 4
        bne     $t1, $a3, inner
        addi    $t0, $t0, 4
        bne     $t0, $a2, outer

```

2.31 [10] <§§2.3, 2.6, 2.9> Assume that the code from Exercise 2.30 is run on a machine with a 2 GHz clock that requires the following number of cycles for each instruction:

Instruction	Cycles
add, addi, sll	1
lw, bne	2

In the worst case, how many seconds will it take to execute this code?

2.32 [5] <§2.9> Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
b = 25 | a;
```

Assume that a corresponds to register \$t0 and b corresponds to register \$t1.

2.33 [10] <§2.9>  **For More Practice:** Translating from C to MIPS

2.34 [10] <§§ 2.3, 2.6, 2.9> The following program tries to copy words from the address in register \$a0 to the address in register \$a1, counting the number of words copied in register \$v0. The program stops copying when it finds a word equal to 0. You do not have to preserve the contents of registers \$v1, \$a0, and \$a1. This terminating word should be copied but not counted.

```

        addi    $v0, $zero, 0 # Initialize count
loop:   lw      $v1, 0($a0)    # Read next word from source
        sw      $v1, 0($a1)    # Write to destination
        addi    $a0, $a0, 4    # Advance pointer to next source
        addi    $a1, $a1, 4    # Advance pointer to next destination

```



```
beq $v1, $zero, loop # Loop if word copied != zero
```

There are multiple bugs in this MIPS program; fix them and turn in a bug-free version. Like many of the exercises in this chapter, the easiest way to write MIPS programs is to use the simulator described in [Appendix A](#).

2.35 [10] <§§2.2, 2.3, 2.6, 2.9> [For More Practice](#): Reverse Translation from MIPS to C

2.36 <§2.9> [For More Practice](#): Translating from C to MIPS

2.37 [25] <§2.10> As discussed on page 107 (Section 2.10, “Assembler”), pseudoinstructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudoinstruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences. In the following table, `big` refers to a specific number that requires 32 bits to represent and `small` to a number that can fit in 16 bits.

Pseudoinstruction	What it accomplishes
<code>move \$t1, \$t2</code>	<code>\$t1 = \$t2</code>
<code>clear \$t50</code>	<code>\$t0 = 0</code>
<code>beq \$t1, small, L</code>	if (<code>\$t1 = small</code>) go to L
<code>beq \$t2, big, L</code>	if (<code>\$t2 = big</code>) go to L
<code>li \$t1, small</code>	<code>\$t1 = small</code>
<code>li \$t2, big</code>	<code>\$t2 = big</code>
<code>ble \$t3, \$t5, L</code>	if (<code>\$t3 <= \$t5</code>) go to L
<code>bgt \$t4, \$t5, L</code>	if (<code>\$t4 > \$t5</code>) go to L
<code>bge \$t5, \$t3, L</code>	if (<code>\$t5 >= \$t3</code>) go to L
<code>addi \$t0, \$t2, big</code>	<code>\$t0 = \$t2 + big</code>
<code>lw \$t5, big(\$t2)</code>	<code>\$t5 = Memory[\$t2 + big]</code>

2.38 [5] <§§2.9, 2.10> Given your understanding of PC-relative addressing, explain why an assembler might have problems directly implementing the branch instruction in the following code sequence:

```
here:                beq    $s0, $s2, there
...
there                add    $s0, $s0, $s0
```

Show how the assembler might rewrite this code sequence to solve these problems.

2.39 <§2.10> [For More Practice](#): MIPS Pseudoinstructions

2.40 <§2.10> [For More Practice](#): Linking MIPS Code

2.41 <§2.10>  **For More Practice:** Linking MIPS Code

2.42 [20] <§2.11> Find a large program written in C (for example, gcc, which can be obtained from <http://gcc.gnu.org>) and compile the program twice, once with optimizations (use -O3) and once without. Compare the compilation time and run time of the program. Are the results what you expect?

2.43 [20] <§2.12>  **For More Practice:** Enhancing MIPS Addressing Modes

2.44 [10] <§2.12>  **For More Practice:** Enhancing MIPS Addressing Modes

2.45 [10] <§2.12>  **In More Depth:** The IBM/Motorola versus MIPS in C

2.46 [15] <§§2.6, 2.13> The MIPS translation of the C (or Java) segment

```
while (save[i] == k)
    i += 1;
```

on page 129 (Section 2.6, “Compiling a While Loop in C”) uses both a conditional branch and an unconditional jump each time through the loop. Only poor compilers would produce code with this loop overhead. Assuming that this code is in Java (not C), rewrite the assembly code so that it uses at most one branch or jump each time through the loop. Additionally, add code to perform the Java checking for index out of bounds and ensure that this code uses at most one branch or jump each time through the loop. How many instructions are executed before and after the optimization if the number of iterations of the loop is 10 and the value of *i* is never out of bounds?

2.47 [30] <§§2.6, 2.13> Consider the following fragment of Java code:

```
for (i=0; i<=100; i=i+1)
    a[i] = b[i] + c;
```

Assume that *a* and *b* are arrays of words and the base address of *a* is in \$a0 and the base address of *b* is in \$a1. Register \$t0 is associated with variable *i* and register \$s0 with the value of *c*. You may also assume that any address constants you need are available to be loaded from memory. Write the code for MIPS. How many instructions are executed during the running of this code if there are no array out-of-bounds exceptions thrown? How many memory data references will be made during execution?

2.48 [5] <§2.13> Write the MIPS code for the Java method `compareTo` (found in Figure 2.35 on page 124).

2.49 [15] <§2.17> When designing memory systems, it becomes useful to know the frequency of memory reads versus writes as well as the frequency of accesses for instructions versus data. Using the average instruction mix information for MIPS for the program SPEC2000int in Figure 2.48 (on page 141), find the following:

- a. The percentage of all memory accesses (both data and instruction) that are for data.
- b. The percentage of all memory accesses (both data and instruction) that are for reads. Assume that two-thirds of data transfers are loads.

2.50 [10] <§2.17> Perform the same calculations as for Exercise 2.49, but replace the program SPEC2000int with SPEC2000fp.

2.51 [15] <§2.17> Suppose we have made the following measurements of average CPI for instructions:

Instruction	Average CPI
Arithmetic	1.0 clock cycles
Data transfer	1.4 clock cycles
Conditional branch	1.7 clock cycles
Jump	1.2 clock cycles

Compute the effective CPI for MIPS. Average the instruction frequencies for SPEC2000int and SPEC2000fp in Figure 2.48 on page 146 to obtain the instruction mix.

2.52 [20] <§2.18>  [In More Depth](#): Instruction Set Styles

2.53 [20] <§2.18>  [In More Depth](#): Instruction Set Styles

2.54 [10] <§2.18>  [In More Depth](#): The Single Instruction Computer

2.55 [20] <§2.18>  [In More Depth](#): The Single Instruction Computer

2.56 [5] <§2.19>The stored-program concept, introduced in the late 1940s, brought about a significant change in how computers were designed and operated. What is a possible example of a nonstored-program machine, and what are the problems with such a machine? How can these problems be overcome by a stored-program machine?

2.57 [5] <§2.19>  [In More Depth](#): The IBM/Motorola versus MIPS in C

2.58 [15] <§2.19>  [In More Depth](#): The IBM/Motorola versus MIPS in C

2.59 [15] <§2.19>  [In More Depth](#): Logical Instructions

Computers in the Real World

Helping Save Our Environment with Data

Problem to solve: Monitor plants and animals of our environment to collect information that may influence environmental policies.

Solution: Develop rugged, battery-operated, embedded computers with sensors, wireless communication, and appropriate software.

Stanford biologist Barbara Block studies bluefin tuna. One policy question was whether the tuna on one side of the Atlantic are different from those on the other side. If so, then each region could set its own quotas. If not, then we need oceanwide quotas.

To answer this question, she started implanting tuna with devices that could monitor their journeys. Every two minutes a pop-up satellite archival tag (PSAT) records water pressure, ambient light, temperature, time of day, and

other measurements. Data are saved in 1 MB of flash memory. The onboard 8-bit microprocessor estimates depth from the water pressure. It finds longitude using light intensity data and time of day. It determines sunrise, sunset, and therefore high noon, and calculates the time shift between local noon and Greenwich Mean Time noon, like a navigator using a sextant and chronometer. The water temperature is later matched to satellite records to determine latitude. Block does not rely on fishermen to catch the tuna and return PSATs. A PSAT is attached to a fish with a pin that dissolves via electrolysis after the computer turns on a battery. The tag then floats to the surface and begins transmitting data to satellites. The floating tag can transmit for up to two weeks, sending the data directly to Block's lab.



Block and students tag a bluefin tuna, which can grow to 2000 pounds and 10 feet in length.



A pop-up archival satellite tag and internal electronics.

Block discovered that bluefin tuna travel more than 10,000 miles per year; tuna tagged near the East Coast of the United States will cross the Atlantic and spawn in both the Gulf of Mexico and the Eastern Mediterranean. Her discovery changed regulations so that tuna are no longer managed separately in the Eastern and Western Atlantic. She is now developing a census of Pacific marine life using smaller tags for smaller animals and tags that transmit each time a fish surfaces. She speculates that tagged tuna could be ideal “vehicles” to monitor ocean change.

Berkeley biologist Todd Dawson studies the ecology of the coastal redwood, *Sequoia sempervirens*, particularly the interaction of sea fog with trees. For years his research involved installing 50 kilograms of gear and kilometers of wire strung to sensors. This work is often done more than 80 meters above the ground. Data could only be retrieved by climbing up to a printer-sized data logger.

Berkeley computer scientist David Culler proposed a new approach. Dawson is now placing miniature wireless sensors the size of film

canisters in these trees. Each micromote is less than 3 cubic inches, can transmit up to 40 KB/sec, and can run for months on a C battery. Since micromotes are small and cheap, many can be placed in a tree. Data is collected with a compatible laptop by simply walking to the base of the tree.

Dawson found that summertime fog accounts for 25% to 40% of the water that the redwoods receive for the whole year. The trees may even be drinking water directly from fog via a symbiotic relationship with fungi living on their leaves.

Dawson predicts wireless sensor networks will change the way people do ecological research.

To learn more see these references on the  library

Block et al., “Migratory movements, depth preferences, and thermal biology of atlantic bluefin tuna,” *Science* 293: 1310–14, 2001

“Redwoods,” Prof. Dawson’s laboratory site

“Redwood’s drinking water from fog,” *The Forestry Source*, Nov. 2002

“Tagging of the Pacific Pelagics,” www.toppcensus.org



Professor Dawson and student climbing a sequoia to install fog monitors.




The Mica micromote with C battery. It is about the size of a film canister.

3

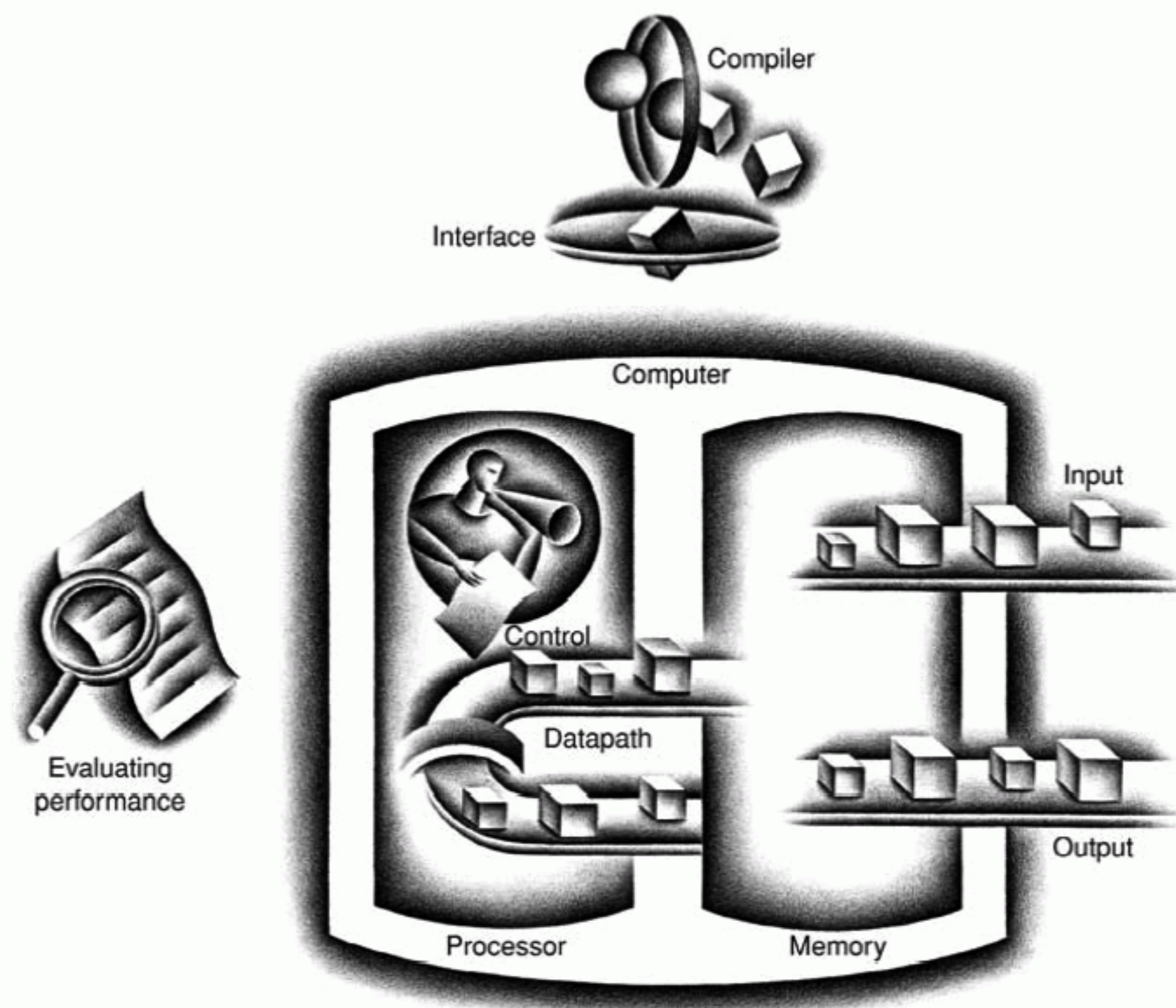
Arithmetic for Computers

*Numerical precision
is the very soul
of science.*

Sir D'arcy Wentworth Thompson
On Growth and Form, 1917

3.1	Introduction	160
3.2	Signed and Unsigned Numbers	160
3.3	Addition and Subtraction	170
3.4	Multiplication	176
3.5	Division	183
3.6	Floating Point	189
3.7	Real Stuff: Floating Point in the IA-32	217
3.8	Fallacies and Pitfalls	220
3.9	Concluding Remarks	225
 3.10	Historical Perspective and Further Reading	229
3.11	Exercises	229

The Five Classic Components of a Computer



3.1 Introduction

Computer words are composed of bits; thus words can be represented as binary numbers. Although the natural numbers 0, 1, 2, and so on can be represented either in decimal or binary form, what about the other numbers that commonly occur? For example:

- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if an operation creates a number bigger than can be represented?
- What about fractions and real numbers?

And underlying all these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel this mystery, including representation of numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may even explain quirks that you have already encountered with computers. (If you are familiar with signed binary numbers, you may wish to skip the next section and go to Section 3.3 on page 170.)

3.2 Signed and Unsigned Numbers

Numbers can be represented in any base; humans prefer base 10 and, as we examined in Chapter 2, base 2 is best for computers. To avoid confusion we subscript decimal numbers with *ten* and binary numbers with *two*.

In any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^i$$

where *i* starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: Simply use the power of the base for that bit. For example,

$$1011_{\text{two}}$$

represents

$$\begin{aligned}
 & (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{ten}} \\
 &= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{ten}} \\
 &= 8 + 0 + 2 + 1_{\text{ten}} \\
 &= 11_{\text{ten}}
 \end{aligned}$$

Hence the bits are numbered 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two} :

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase **least significant bit** is used to refer to the rightmost bit (bit 0 above) and **most significant bit** to the leftmost bit (bit 31).

The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4,294,967,295_{\text{ten}}$):

$$\begin{aligned}
 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000_{\text{two}} &= 0_{\text{ten}} \\
 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_{\text{two}} &= 1_{\text{ten}} \\
 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0010_{\text{two}} &= 2_{\text{ten}} \\
 &\dots \\
 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1101_{\text{two}} &= 4,294,967,293_{\text{ten}} \\
 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_{\text{two}} &= 4,294,967,294_{\text{ten}} \\
 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111_{\text{two}} &= 4,294,967,295_{\text{ten}}
 \end{aligned}$$

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here x_i means the i th bit of x):

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted to all binary, converting to base 10 only for the relatively infrequent input/output events.

least significant bit The rightmost bit in a MIPS word.

most significant bit The leftmost bit in a MIPS word.

Hardware Software Interface

EXAMPLE**ANSWER****ASCII versus Binary Numbers**

We could represent numbers as strings of ASCII digits instead of as integers (see Figure 2.21 on page 91). How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

One billion is 1 000 000 000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be $(10 \times 8)/32$ or 2.5. In addition to the expansion in storage, the hardware to add, subtract, multiply, and divide such numbers is difficult. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the operating system and program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

0000 0000 0000 0000 0000 0000 0000 0000	$_{\text{two}}$	=	0_{ten}
0000 0000 0000 0000 0000 0000 0000 0001	$_{\text{two}}$	=	1_{ten}
0000 0000 0000 0000 0000 0000 0000 0010	$_{\text{two}}$	=	2_{ten}
...	...		
0111 1111 1111 1111 1111 1111 1111 1101	$_{\text{two}}$	=	$2,147,483,645_{\text{ten}}$
0111 1111 1111 1111 1111 1111 1111 1110	$_{\text{two}}$	=	$2,147,483,646_{\text{ten}}$
0111 1111 1111 1111 1111 1111 1111 1111	$_{\text{two}}$	=	$2,147,483,647_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0000	$_{\text{two}}$	=	$-2,147,483,648_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0001	$_{\text{two}}$	=	$-2,147,483,647_{\text{ten}}$
1000 0000 0000 0000 0000 0000 0000 0010	$_{\text{two}}$	=	$-2,147,483,646_{\text{ten}}$
...	...		
1111 1111 1111 1111 1111 1111 1111 1101	$_{\text{two}}$	=	-3_{ten}
1111 1111 1111 1111 1111 1111 1111 1110	$_{\text{two}}$	=	-2_{ten}
1111 1111 1111 1111 1111 1111 1111 1111	$_{\text{two}}$	=	-1_{ten}

The positive half of the numbers, from 0 to $2,147,483,647_{\text{ten}}$ ($2^{31} - 1$), use the same representation as before. The following bit pattern ($1000 \dots 0000_{\text{two}}$) represents the most negative number $-2,147,483,648_{\text{ten}}$ (-2^{31}). It is followed by a declining set of negative numbers: $-2,147,483,647_{\text{ten}}$ ($1000 \dots 0001_{\text{two}}$) down to -1_{ten} ($1111 \dots 1111_{\text{two}}$).

Two's complement does have one negative number, $-2,147,483,648_{\text{ten}}$, that has no corresponding positive number. Such imbalance was a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with 0 considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

The sign bit is multiplied by -2^{31} , and the rest of the bits are then multiplied by positive versions of their respective base values.

EXAMPLE**Binary to Decimal Conversion**

What is the decimal value of this 32-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1100_{two}

ANSWER

Substituting the number's bit values into the formula above:

$$\begin{aligned}
 & (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \\
 &= -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\
 &= -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}} \\
 &= -4_{\text{ten}}
 \end{aligned}$$

We'll see a shortcut to simplify conversion soon.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (l**b**) treats the byte as a signed number and thus sign-extends to fill the 24 leftmost bits of the register, while *load byte unsigned* (l**b**u) works with unsigned integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, l**b**u is used practically exclusively for byte loads. For similar reasons, *load half* (l**h**) treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register, while *load halfword unsigned* (l**h**u) works with unsigned integers.

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former *integers* (declared as `int` in the program) and the latter *unsigned integers* (unsigned `int`). Some C style guides even recommend declaring the former as signed `int` to keep the distinction clear.

Comparison instructions must deal with this dichotomy. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking in a few pages.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

Hardware Software Interface

Signed versus Unsigned Comparison

Suppose register `$s0` has the binary number

1111 1111 1111 1111 1111 1111 1111 1111_{two}

and that register `$s1` has the binary number

0000 0000 0000 0000 0000 0000 0000 0001_{two}

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt      $t0, $s0, $s1 # signed comparison
sltu     $t1, $s0, $s1 # unsigned comparison
```

EXAMPLE

ANSWER

The value in register \$s0 represents -1 if it is an integer and $4,294,967,295_{\text{ten}}$ if it is an unsigned integer. The value in register \$s1 represents 1 in either case. Then register \$t0 has the value 1, since $-1_{\text{ten}} < 1_{\text{ten}}$, and register \$t1 has the value 0, since $4,294,967,295_{\text{ten}} > 1_{\text{ten}}$.

Before going on to addition and subtraction, let's examine a few useful shortcuts when working with two's complement numbers.

The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111 \dots 111_{\text{two}}$, which represents -1 . Since $x + x \equiv -1$, therefore $x + x + 1 = 0$ or $x + 1 = -x$.

EXAMPLE**ANSWER****Negation Shortcut**

Negate 2_{ten} , and then check the result by negating -2_{ten} .

$2_{\text{ten}} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}}$
Negating this number by inverting the bits and adding one,

$$\begin{array}{r}
 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{two}} \\
 + \qquad \qquad \qquad 1_{\text{two}} \\
 \hline
 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} \\
 = -2_{\text{ten}}
 \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

is first inverted and then incremented:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} \\
 + \qquad \qquad \qquad 1_{\text{two}} \\
 \hline
 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} \\
 = 2_{\text{ten}}
 \end{array}$$

The second shortcut tells us how to convert a binary number represented in n bits to a number represented with more than n bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32,768_{\text{ten}}$ (-2^{15}) to $32,767_{\text{ten}}$ ($2^{15} - 1$). To add the immediate field to a 32-bit register, the computer must convert that 16-bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension*.

Sign Extension Shortcut

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

The 16-bit binary version of the number 2 is

$$0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}$$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000\ 0000\ 0000\ 0010_{\text{two}}$$

becomes

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_{\text{two}} \\ + \ 1_{\text{two}} \\ \hline = 1111\ 1111\ 1111\ 1110_{\text{two}} \end{array}$$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and those that are negative two's complement

EXAMPLE

ANSWER

numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

The third shortcut reduces the cost of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative.

EXAMPLE

ANSWER

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$a1 ≥ $t2` or if `$a1` is negative.

The checking code just uses `sltu` to do both checks:

```
sltu $t0,$a1,$t2 # Temp reg $t0=0 if k>=length or k<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the overwhelming choice since 1965 has been two's complement. Figure 3.1 shows the additions to the MIPS assembly language revealed in this section. (The MIPS machine language is also illustrated on the back endpapers of this book.)

Check Yourself

Which type of variable that can contain $1,000,000,000_{\text{ten}}$ takes the most memory space?

1. `int` in C
2. `string` in C
3. `string` in Java (which uses Unicode)

MIPS operands

Name	Example	Comments
32 registers	\$s0–\$s7, \$t0–\$t9, \$gp, \$fp, \$zero, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants.
2 ³⁰ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three operands
	add immediate	addi \$s1,\$s2,100	\$s1 = \$s2 + 100	+ constant
Data transfer	load word	lw \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Word from memory to register
	store word	sw \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Word from register to memory
	load half unsigned	lhu \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Halfword memory to register
	store half	sh \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Halfword register to memory
	load byte unsigned	lbu \$s1,100(\$s2)	\$s1 = Memory[\$s2 + 100]	Byte from memory to register
	store byte	sb \$s1,100(\$s2)	Memory[\$s2 + 100] = \$s1	Byte from register to memory
	load upper immediate	lui \$s1,100	\$s1 = 100 * 2 ¹⁶	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~ (\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	\$s1 = \$s2 & 100	Bit-by-bit AND with constant
	or immediate	ori \$s1,\$s2,100	\$s1 = \$s2 100	Bit-by-bit OR with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; two's complement
	set less than immediate	slti \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compare < constant; two's complement
	set less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; unsigned numbers
	set less than immediate unsigned	sltiu \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compare < constant; unsigned numbers
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

FIGURE 3.1 MIPS architecture revealed thus far. Color indicates portions from this section added to the MIPS architecture revealed in Chapter 2 (Figure 3.26 on page 228). MIPS machine language is listed in the MIPS summary reference card in the front of this book.

biased notation A notation that represents the most negative value by $00 \dots 000_{\text{two}}$ and the most positive value by $11 \dots 11_{\text{two}}$, with 0 typically having the value $10 \dots 00_{\text{two}}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

Subtraction: Addition's Tricky Pal

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., *Book of Top Ten Lists*, 1990

EXAMPLE

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n -bit number and its negative is 2^n ; hence, the complement or negation of a two's complement number x is $2^n - x$.

A third alternative representation is called *one's complement*. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, which helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00 \dots 00_{\text{two}}$ is positive 0 and $11 \dots 11_{\text{two}}$ is negative 0. The most negative number $10 \dots 000_{\text{two}}$ represents $-2,147,483,647_{\text{ten}}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point, is to represent the most negative value by $00 \dots 000_{\text{two}}$ and the most positive value represented by $11 \dots 11_{\text{two}}$, with 0 typically having the value $10 \dots 00_{\text{two}}$. This is called a **biased notation**, since it biases the number such that the number plus the bias has a nonnegative representation.

Elaboration: For signed decimal numbers we used “−” to represent negative because there are no limits to the size of a decimal number. Given a fixed word size, binary and hexadecimal bit strings can encode the sign, and hence we do not normally use “+” or “−” with binary or hexadecimal notation.

3.3

Addition and Subtraction

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: The appropriate operand is simply negated before being added.

Binary Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 + \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}}
 \end{array}$$

ANSWER

The 4 bits to the right have all the action; Figure 3.2 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

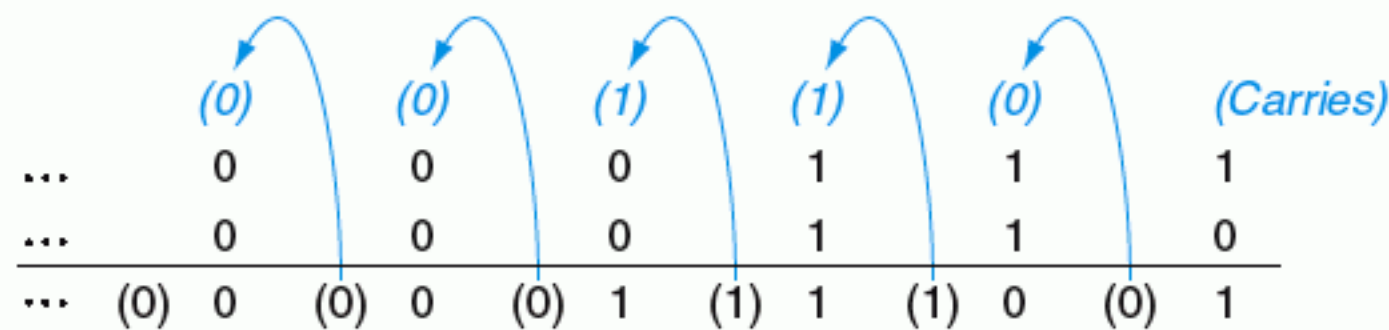


FIGURE 3.2 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is $0 + 1 + 1$. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of $1 + 1 + 1$, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is $1 + 0 + 0$, yielding a 1 sum and no carry.

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 - \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

or via addition using the two's complement representation of -6 :

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 + \quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

We said earlier that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, $-10 + 4 = -6$. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: When the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $x - y = x + (-y)$ because we subtract by negating the second operand and then add. So, when we subtract operands of

the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Having examined when overflow cannot occur in addition and subtraction, we still haven't answered how to detect when it *does* occur. Overflow occurs when adding two positive numbers and the sum is negative, or vice versa. Clearly, adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs the sign bit is being set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. This means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means a borrow occurred from the sign bit. Figure 3.3 shows the combination of operations, operands, and results that indicate an overflow.

We have just seen how to detect overflow for two's complement numbers in a computer. What about unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
- Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions addu, addiu, and subu no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.3 Overflow conditions for addition and subtraction.

The computer designer must decide how to handle arithmetic overflows. Although some languages like C ignore integer overflow, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.

MIPS detects overflow with an **exception**, also called an **interrupt** on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 5.6 covers exceptions in more detail; Chapters 7 and 8 describe other situations where exceptions and interrupts occur.)

MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

Hardware Software Interface

exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Elaboration: MIPS can trap on overflow, but unlike many other computers there is no conditional branch to test overflow. A sequence of MIPS instructions can discover overflow. For signed addition, the sequence is the following (see the In More Depth segment on logical instruction in Chapter 2 for the definition of the `xor` instructions):

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
so no overflow
xor  $t3, $t0, $t1 # signs =; sign of sum match too?
                        # $t3 negative if sum sign different
slt  $t3, $t3, $zero # $t3 = 1 if sum sign different
bne  $t3, $zero, Overflow # All three signs ≠; go to
overflow
```


For unsigned addition ($\$t0 = \$t1 + \$t2$), the test is

```
addu $t0, $t1, $t2 # $t0 = sum
nor  $t3, $t1, $zero # $t3 = NOT $t1
      # (2's comp - 1:  $2^{32} - \$t1 - 1$ )
sltu $t3, $t3, $t2 # ( $2^{32} - \$t1 - 1$ ) < $t2
      #  $\Rightarrow 2^{32} - 1 < \$t1 + \$t2$ 
bne $t3, $zero, Overflow # if( $2^{32}-1 < \$t1 + \$t2$ ) go to
overflow
```

Summary

The main point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require detection of overflow, so today all computers have a way to detect it. Figure 3.4 shows the additions to the MIPS architecture from this section.

Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half. What MIPS instructions would be used?

1. Load with `lbu`, `lhu`; arithmetic with `add`, `sub`, `mult`, `div`; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mult`, `div`; then store using `sb`, `sh`.
3. Loads with `lb`, `lh`; arithmetic with `add`, `sub`, `mult`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.

Elaboration: In the preceding text, we said that you copy EPC into a register via `mfc0` and then return to the interrupted code via jump register. This leads to an interesting question: Since you must first transfer EPC to a register to use with jump register, how can jump register return to the interrupted code *and* restore the original values of *all* registers? You either restore the old registers first, thereby destroying your return address from EPC that you placed in a register for use in jump register, or you restore all registers but the one with the return address so that you can jump—meaning an exception would result in changing that one register at any time during program execution! Neither option is satisfactory.

To rescue the hardware from this dilemma, MIPS programmers agreed to reserve registers `$k0` and `$k1` for the operating system; these registers are *not* restored on exceptions. Just as the MIPS compilers avoid using register `$at` so that the assembler

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Used to copy Exception PC plus other special registers
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load half unsigned	lhu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Halfword memory to register
	store half	sh \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Halfword register to memory
	load byte unsigned	lbu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$	Bit-by-bit OR with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 \neq \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; two's complement
	set less than immediate	slti \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; two's complement
	set less than unsign	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; unsigned
	set less than immediate unsigned	sltiu \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

FIGURE 3.4 MIPS architecture revealed thus far. To save space in the table, it does not include the registers and memory found in Figure 3.1 on page 169. Color indicates the portions revealed since Figure 3.1. MIPS machine language is also on the MIPS reference summary card.

Multiplication is vexation,
Division is as bad;
The rule of three doth puzzle
me,
And practice drives me mad.
Anonymous, Elizabethan
manuscript, 1570

can use it as a temporary register (see the Hardware Software Interface section on page 96 in Chapter 2), compilers also abstain from using registers \$k0 and \$k1 to make them available for the operating system. Exception routines place the return address in one of these registers and then use jump register to restore the instruction address.

3.4

Multiplication

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiply.

But first let’s review the multiplication of decimal numbers in longhand to remind ourselves of the steps and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. Multiplying 1000_{ten} by 1001_{ten} :

Multiplicand		1000 _{ten}
Multiplier	x	1001 _{ten}

		1000
		0000
		0000
		1000

Product		1001000 _{ten}

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier and shifting the intermediate product one digit to the left of the earlier intermediate products.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an n -bit multiplicand and an m -bit multiplier is a product that is $n + m$ bits long. That is, $n + m$ bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In this example we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

- 1. Just place a copy of the multiplicand ($1 \times \text{multiplicand}$) in the proper place if the multiplier digit is a 1, or
- 2. Place 0 ($0 \times \text{multiplicand}$) in the proper place if the digit is 0.

Although the decimal example above happened to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

Sequential Version of the Multiplication Algorithm and Hardware

This design mimics the algorithm we learned in grammar school; the hardware is shown in Figure 3.5. We have drawn the hardware so that data flows from top to bottom to more closely resemble the paper-and-pencil method.

Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step as it may be added to the intermediate products. Over 32 steps a 32-bit multiplicand would move 32 bits to the left. Hence we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and 0 in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

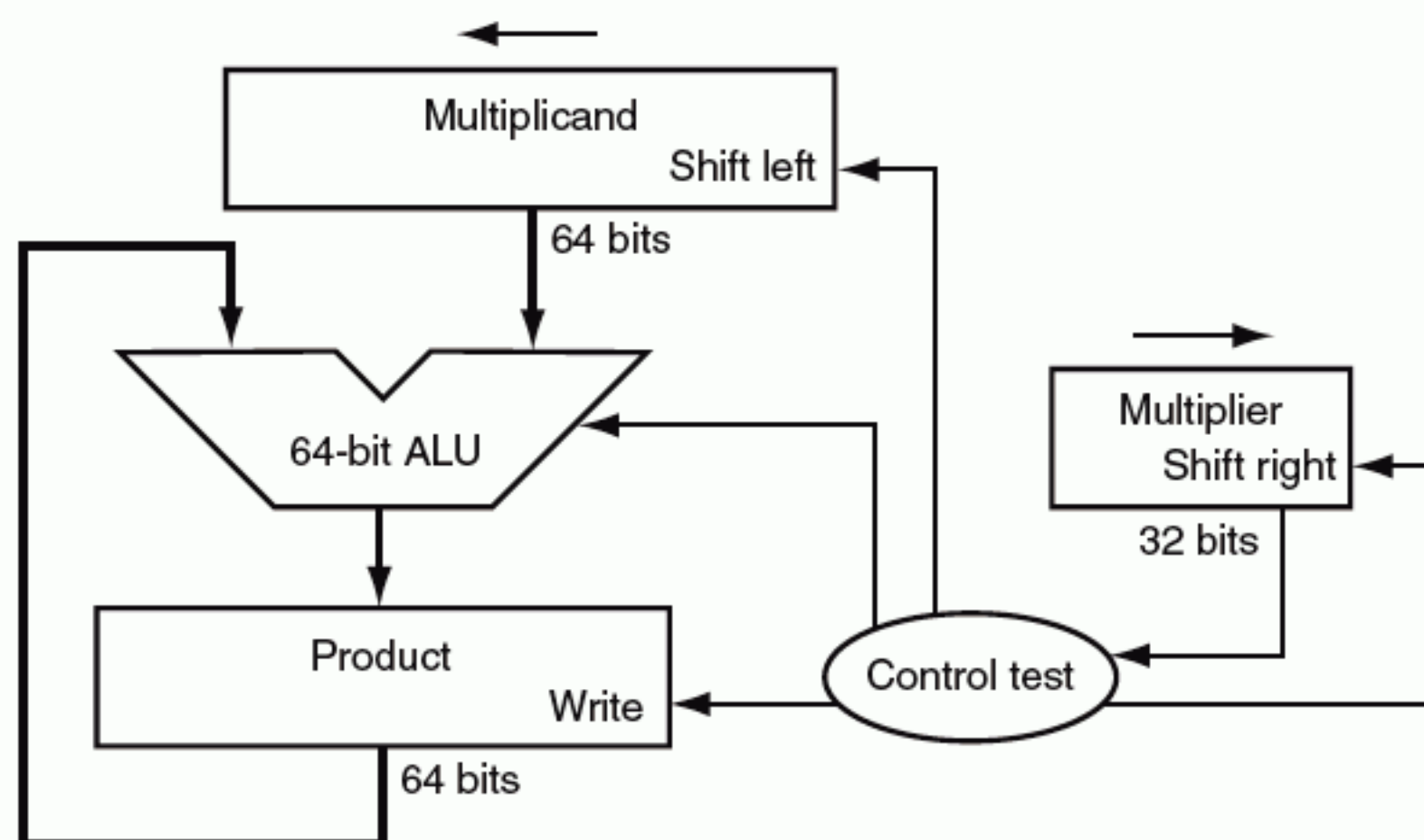


FIGURE 3.5 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Figure 3.6 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product. If each step took a

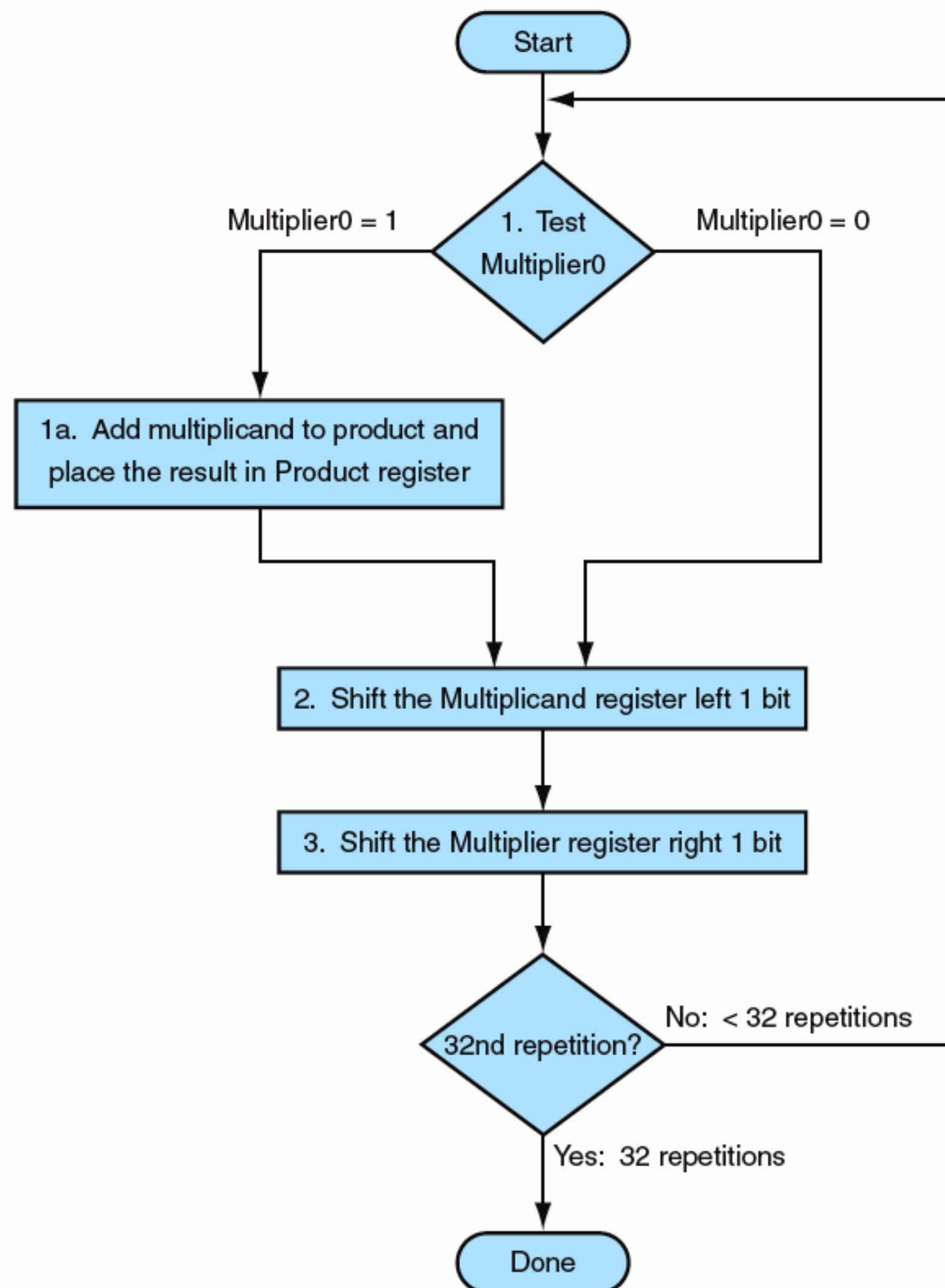


FIGURE 3.6 The first multiplication algorithm, using the hardware shown in Figure 3.5. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's law (see Chapter 4, page 267) reminds us that even a moderate frequency for a slow operation can limit performance.

This algorithm and hardware are easily refined to take 1 clock cycle per step. The speedup comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a one. The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.7 shows the revised hardware.

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. As mentioned in Chapter 2, almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

Hardware Software Interface

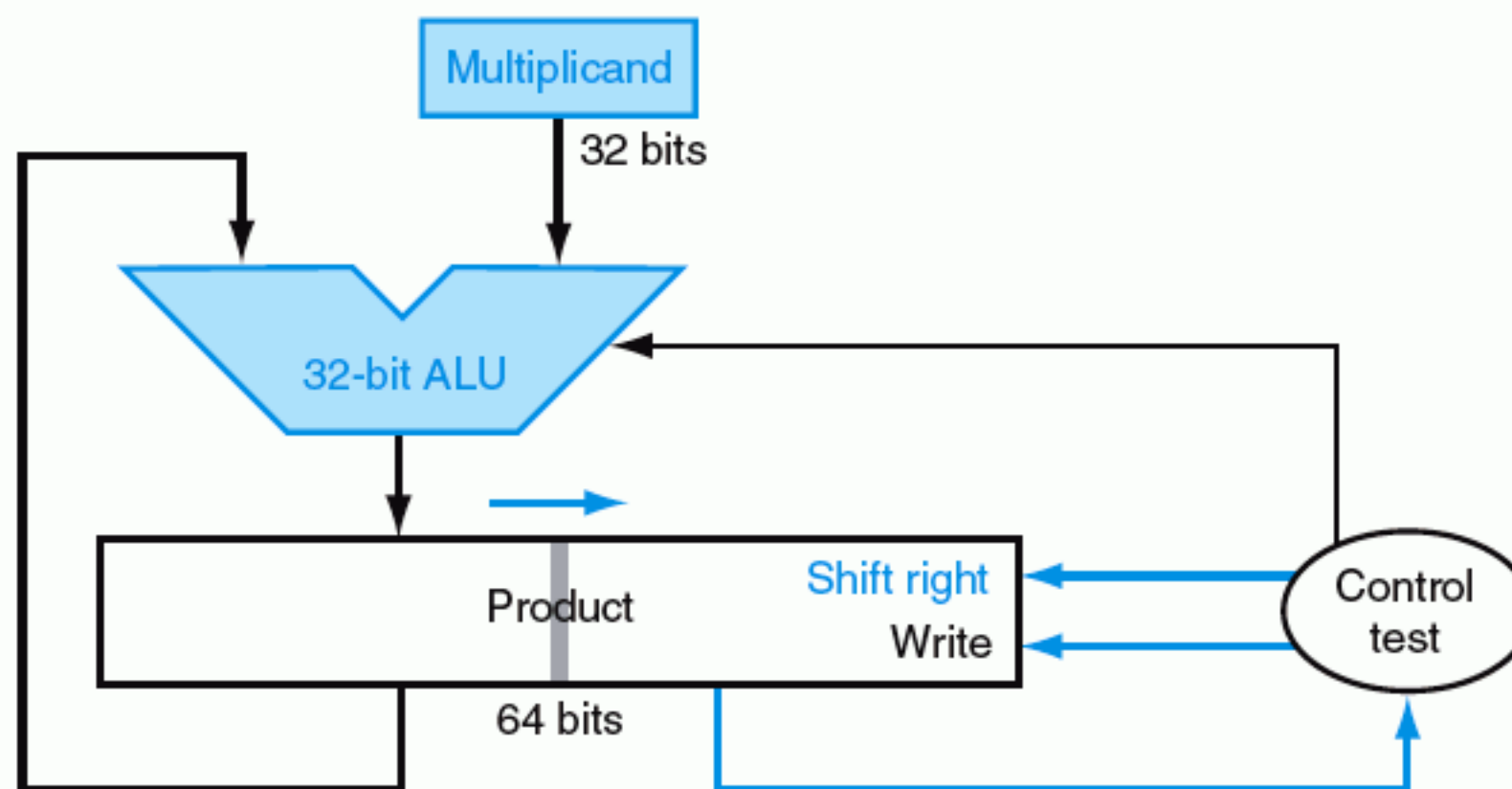


FIGURE 3.7 Refined version of the multiplication hardware. Compare with the first version in Figure 3.5. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color.

EXAMPLE

ANSWER

A Multiply Algorithm

Using 4-bit numbers to save space, multiply $2_{\text{ten}} \times 3_{\text{ten}}$, or $0010_{\text{two}} \times 0011_{\text{two}}$.

Figure 3.8 shows the value of each register for each of the steps labeled according to Figure 3.6, with the final value of $0000\ 0110_{\text{two}}$ or 6_{ten} . Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Signed Multiplication

So far we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

It turns out that the last algorithm will work for signed numbers provided that we remember that the numbers we are dealing with have infinite digits, and that we are only representing them with 32 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011 ¹	0000 0010	0000 0000
1	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001 ¹	0000 0100	0000 0010
2	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000 ⁰	0000 1000	0000 0110
3	1: $0 \Rightarrow$ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000 ⁰	0001 0000	0000 0110
4	1: $0 \Rightarrow$ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.8 Multiply example using algorithm in Figure 3.6. The bit examined to determine the next step is circled in color.

Faster Multiplication

Moore's Law has provided so much more in resources that hardware designers can now build a much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder. Figure 3.9 shows how they would be connected.

Why is this much hardware faster? The sequential multiplier pays the overhead of a clock for each bit of the product. This multiplier array of adders does not. A second reason is this large collection of adders lends itself to many optimizations to gain further improvements. One example is using *carry save adders* to add such a large column of numbers; see Exercises 3.24 and 3.49. A third reason is that it is easy to pipeline such a design to be able to support many multiplies simultaneously (see Chapter 6).

Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*. To produce a properly signed or unsigned product, MIPS has two instructions: multiply (`mult`) and multiply unsigned (`multu`). To fetch the integer 32-bit product, the programmer uses *move from lo* (`mflo`). The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

Summary

Multiplication is accomplished by simple shift and add hardware, derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of two.

Both MIPS multiply instructions ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits. There is no overflow if *Hi* is 0 for `multu` or the replicated sign of *Lo* for `mult`. The instruction *move from hi* (`mfhi`) can be used to transfer *Hi* to a general-purpose register to test for overflow.

**Hardware
Software
Interface**

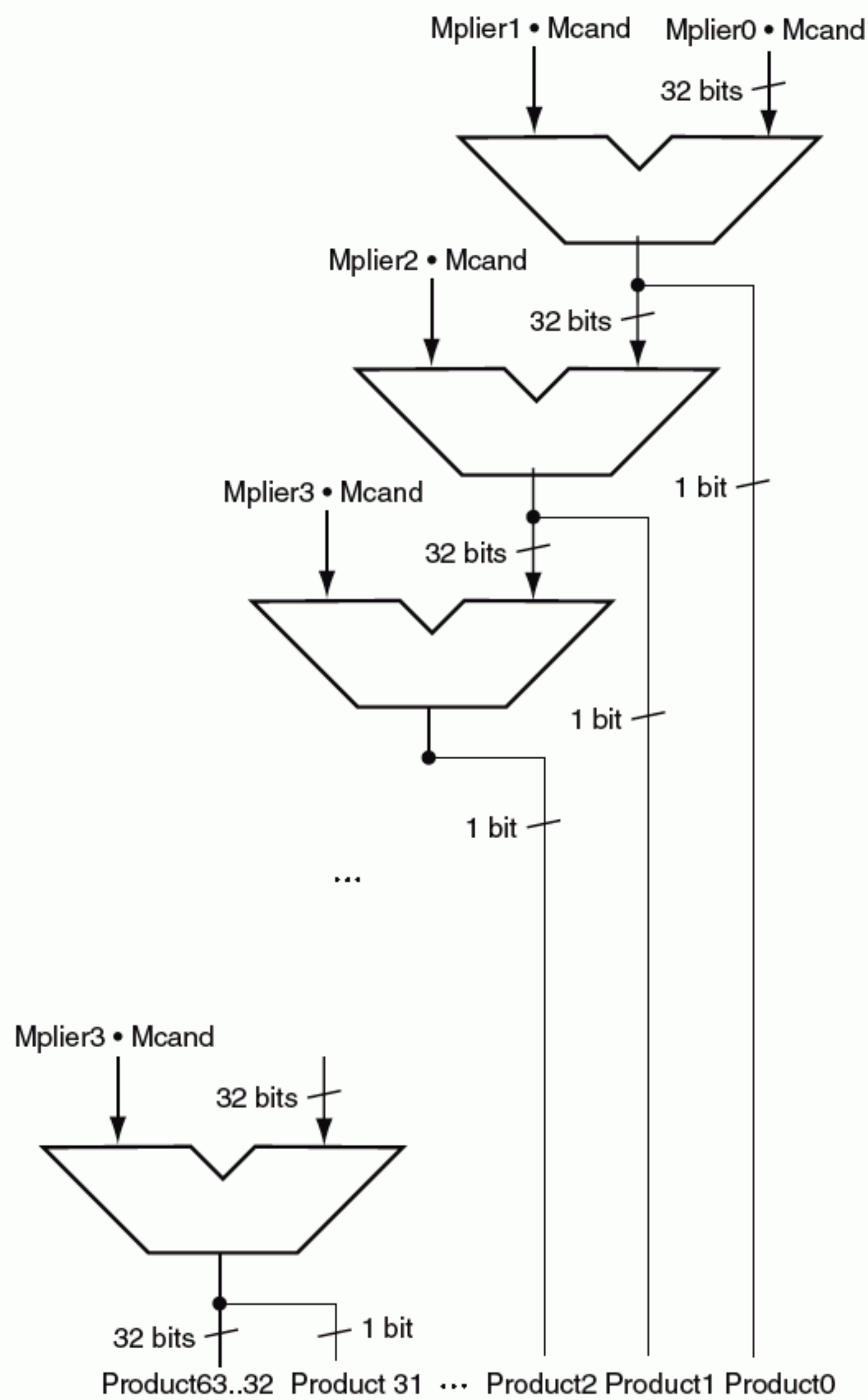


FIGURE 3.9 Fast multiplication hardware. Rather than use a single 32-bit adder 32 times, this hardware “unrolls the loop” to use 32 adders. Each adder produces a 32-bit sum and a carry out. The least significant bit is a bit of the product, and the carry out and the upper 31 bits of the sum are passed along to the next adder.

3.5

Division

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let’s start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1,001,010_{\text{ten}}$ by 1000_{ten} :

Divisor 1000_{ten}

1001_{ten}

1001010_{ten}

-1000

10

101

1010

-1000

10_{ten}

Quotient

Dividend

Remainder

The two operands (**dividend** and **divisor**) and the result (**quotient**) of divide are accompanied by a second result called the **remainder**. Here is another way to express the relationship between the components:

Dividend = Quotient × Divisor + Remainder

where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient.

The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses only the numbers 0 and 1, so it’s easy to figure out how many times the divisor goes into the portion of the dividend: it’s either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

Let’s assume that both the dividend and divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 32-bit values, and we will ignore the sign for now.

Divide et impera.
Latin for “Divide and rule,”
ancient political maxim cited
by Machiavelli, 1532

dividend A number being divided.

divisor A number that the dividend is divided by.

quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

A Division Algorithm and Hardware

Figure 3.10 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

Figure 3.11 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

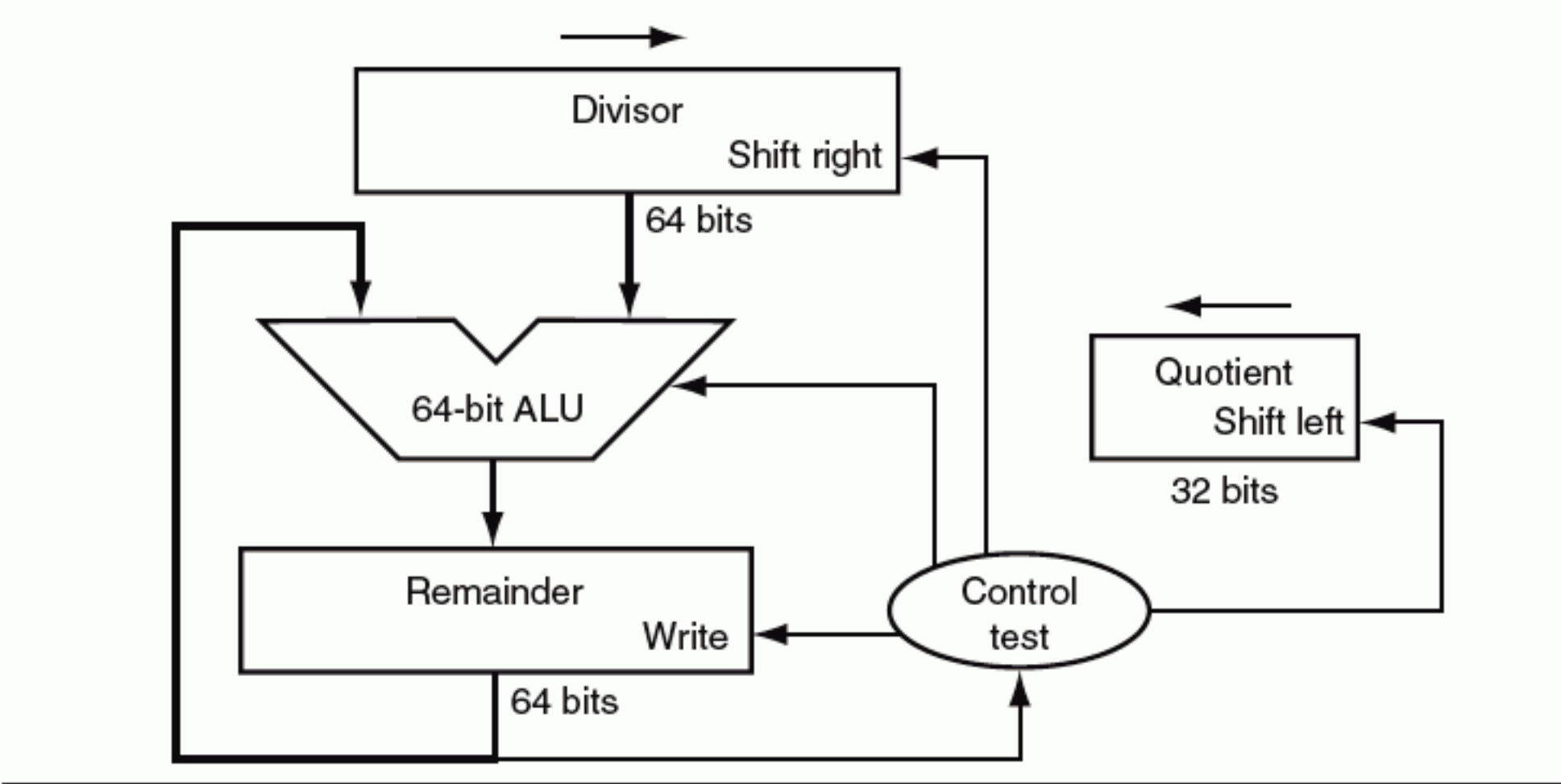


FIGURE 3.10 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

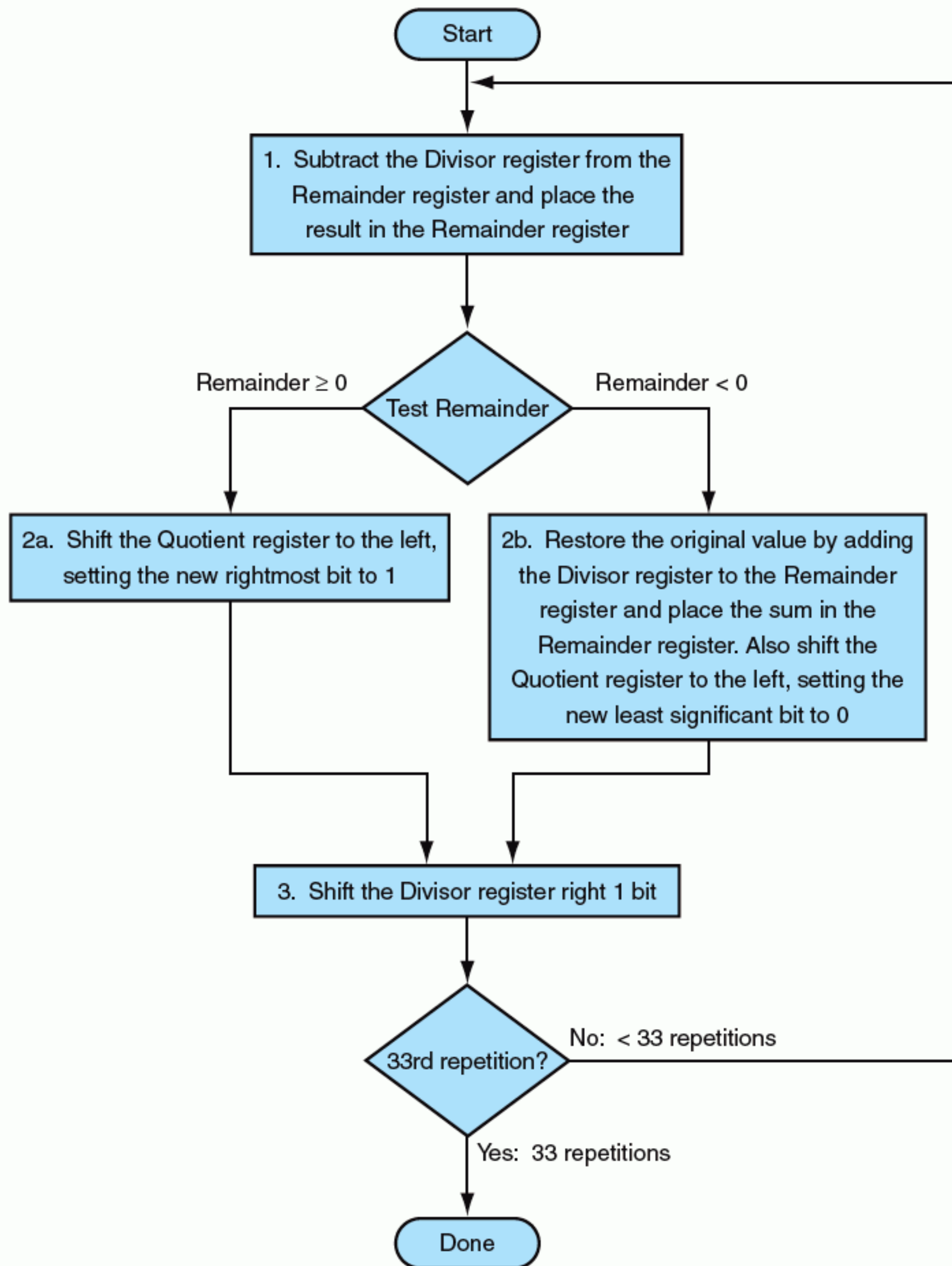


FIGURE 3.11 A division algorithm, using the hardware in Figure 3.10. If the Remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative Remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

EXAMPLE

ANSWER

A Divide Algorithm

Using a 4-bit version of the algorithm to save pages, let’s try dividing 7_{ten} by 2_{ten} , or $0000\ 0111_{\text{two}}$ by 0010_{two} .

Figure 3.12 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten} . Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

This algorithm and hardware can be refined to be faster and cheaper. The speedup comes from shifting the operands and quotient at the same time as the subtract. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.13 shows the revised hardware.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 \Rightarrow +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	①000 0011
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	①000 0001
	2a: Rem \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.12 Division example using the algorithm in Figure 3.11. The bit examined to determine the next step is circled in color.

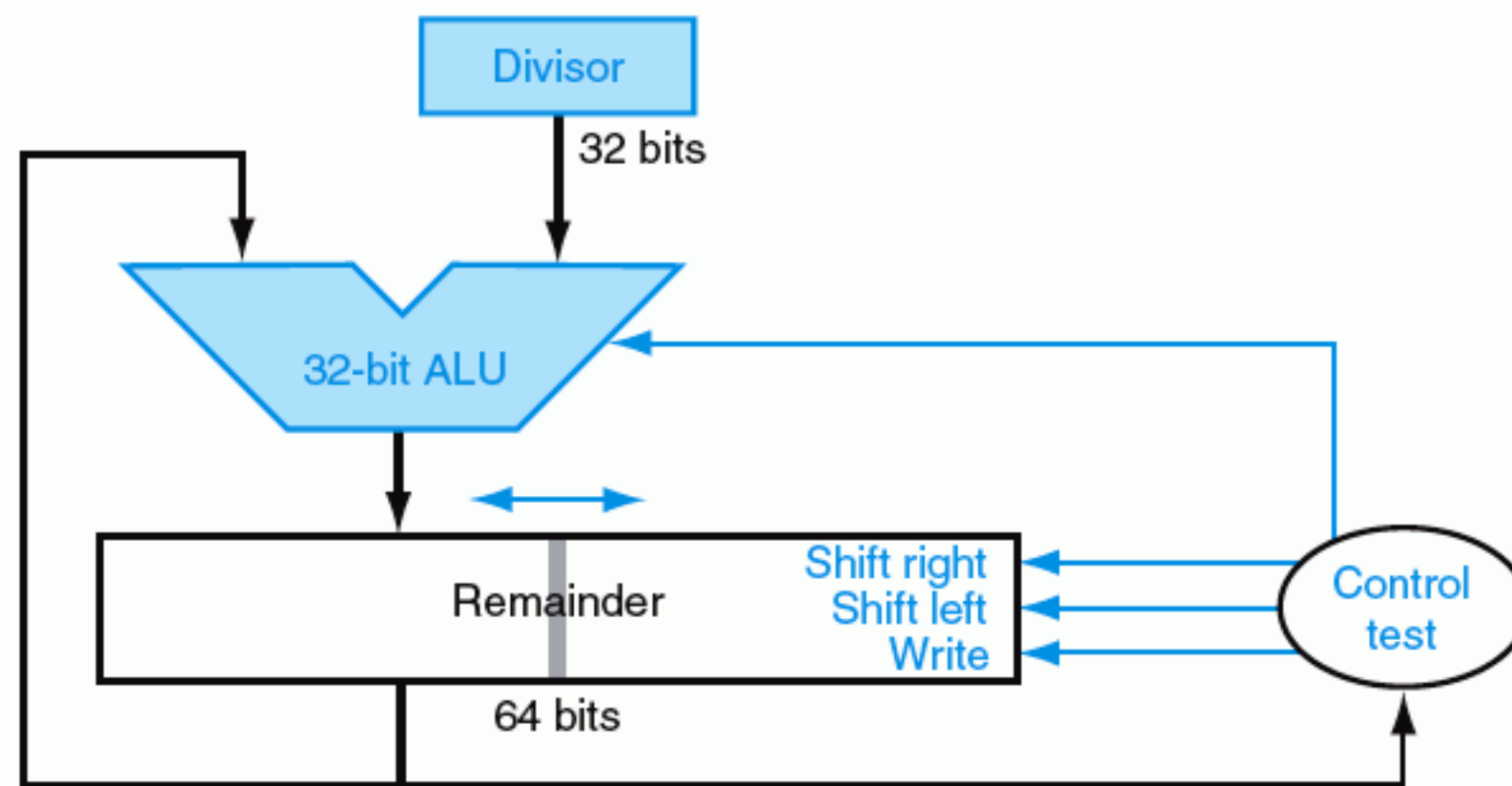


FIGURE 3.13 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.10, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register.

Signed Division

So far we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration: The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{\text{ten}}$ by $\pm 2_{\text{ten}}$. The first case is easy:

$$+7 \div +2: \text{Quotient} = +3, \text{Remainder} = +1$$

Checking the results:

$$7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2: \text{Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\text{Remainder} = (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3 \times +2) = -7 - (-6) = -1$$

So,

$$-7 \div +2: \text{Quotient} = -3, \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of -4 and a remainder of $+1$, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly if

$$-(x \div y) \neq (-x) \div y$$

programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

We calculate the other combinations by following the same rule:

$+7 \div -2$: Quotient = -3 , Remainder = $+1$

$-7 \div -2$: Quotient = $+3$, Remainder = -1

Thus the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster Division

We used 32 adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per bit. The *SRT division* technique tries to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong guesses. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from divisor to index a table that determines the guess for each step.

The accuracy of this fast method depends on having proper values in the lookup table. The fallacy on page 222 in Section 3.8 shows what can happen if the table is incorrect.

Divide in MIPS

You may have already observed that the same sequential hardware can be used for both multiply and divide in Figures 3.7 and 3.13. The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts. Hence, MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide. As we might expect from the algorithm above, Hi contains the remainder, and Lo contains the quotient after the divide instruction completes.

To handle both signed integers and unsigned integers, MIPS has two instructions: *divide* (`div`) and *divide unsigned* (`divu`). The MIPS assembler allows divide instructions to specify three registers, generating the `mflo` or `mfhi` instructions to place the desired result into a general-purpose register.

Summary

The common hardware support for multiply and divide allows MIPS to provide a single pair of 32-bit registers that are used both for multiply and divide. Figure 3.14 summarizes the additions to the MIPS architecture for the last two sections.

MIPS divide instructions ignore overflow, so software must determine if the quotient is too large. In addition to overflow, division can also result in an improper calculation: division by 0. Some computers distinguish these two anomalous events. MIPS software must check the divisor to discover division by 0 as well as overflow.

Hardware Software Interface

Elaboration: An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step since $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes 1 clock per step, is explored further in Exercise 3.29; the algorithm here is called *restoring* division.

3.6

Floating Point

*Speed gets you nowhere if
you're headed the wrong
way.*

American proverb

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265 \dots_{\text{ten}}$ (π)

$2.71828 \dots_{\text{ten}}$ (e)

0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in a nanosecond)

$3,155,760,000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^9$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{\text{ten}} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{\text{ten}} \times 10^{-8}$ and $10.0_{\text{ten}} \times 10^{-10}$ are not.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

normalized A number in floating-point notation that has no leading 0s.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Copy Exception PC + special regs
	multiply	mult \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
	divide unsigned	divu \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = \text{Hi}$	Used to get copy of Hi
	move from Lo	mflo \$s1	$\$s1 = \text{Lo}$	Used to get copy of Lo
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load half unsigned	lhu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Halfword memory to register
	store half	sh \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Halfword register to memory
	load byte unsigned	lbu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	Bit-by-bit OR with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Condi- tional branch	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; two's complement
	set less than immediate	slti \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; two's complement
	set less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; natural numbers
	set less than immediate unsigned	sltiu \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; natural numbers
Uncondi- tional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

FIGURE 3.14 MIPS architecture revealed thus far. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the hi and lo registers to support multiply and divide. Color indicates the portions revealed since Figure 3.4 on page 175. MIPS machine language is listed in the MIPS summary reference card at the front of this book.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Computer arithmetic that supports such numbers is called **floating point** because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

$$1.xxxxxxxxx_{\text{two}} \times 2^{yyyy}$$

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we show the exponent in decimal.)

A standard scientific notation for reals in normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the **fraction** and the size of the **exponent** because a fixed word size means you must take a bit from one to add a bit to the other. This trade-off is between precision and range: Increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from Chapter 2 reminds us, good design demands good compromise.

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number. This representation is called *sign and magnitude*, since the sign has a separate bit from the rest of the number.

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

fraction The value, generally between 0 and 1, placed in the fraction field.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit		8 bits								23 bits																					

In general, floating-point numbers are generally of the form

$(-1)^S \times F \times 2^E$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that MIPS does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{\text{ten}} \times 10^{-38}$ and numbers almost as large as $2.0_{\text{ten}} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, people call this event **underflow**. This situation occurs when the negative exponent is too large to fit in the exponent field.

One way to reduce chances of underflow or overflow is to offer another format that has a larger exponent. In C this number is called *double*, and operations on doubles are called **double precision** floating-point arithmetic; **single precision** floating point is the name of the earlier format.

The representation of a double precision floating-point number takes two MIPS words, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *fraction* is the 52-bit number in the fraction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent											fraction																			
1 bit		11 bits											20 bits																		
fraction (continued)																															
32 bits																															

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

double precision A floating-point value represented in two 32-bit words.

single precision A floating-point value represented in a single 32-bit word.

MIPS double precision allows numbers almost as small as $2.0_{\text{ten}} \times 10^{-308}$ and almost as large as $2.0_{\text{ten}} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the larger significand.

These formats go beyond MIPS. They are part of the *IEEE 754 floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

To pack even more bits into the significand, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or 53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus $00 \dots 00_{\text{two}}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s_1, s_2, s_3, \dots , then the value is

$$(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots) \times 2^E$$

Figure 3.15 shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\infty$ or $-\infty$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

IEEE 754 even has a symbol for the result of invalid operations, such as $0/0$ or subtracting infinity from infinity. This symbol is *NaN*, for *Not a Number*. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when it is convenient.

The designers of IEEE 754 also wanted a floating-point representation that could be easily processed by integer comparisons, especially for sorting. This desire is why the sign is in the most significant bit, allowing a quick test of less than, greater than, or equal to 0. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1–254	anything	1–2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

FIGURE 3.15 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the elaboration on page 217.

Placing the exponent before the significand also simplifies sorting of floating-point numbers using integer comparison instructions, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.

Negative exponents pose a challenge to simplified sorting. If we use two’s complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{\text{two}} \times 2^{-1}$ would be represented as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{\text{two}} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The desirable notation must therefore represent the most negative exponent as $00 \dots 00_{\text{two}}$ and the most positive as $11 \dots 11_{\text{two}}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so -1 is represented by the bit pattern of the value $-1 + 127_{\text{ten}}$, or $126_{\text{ten}} = 0111\ 1110_{\text{two}}$, and $+1$ is represented by $1 + 127$, or $128_{\text{ten}} = 1000\ 0000_{\text{two}}$. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The exponent bias for double precision is 1023.

Elaboration: In an attempt to increase range without removing bits from the significand, some computers before the IEEE 754 standard used a base other than 2. For example, the IBM 360 and 370 mainframe computers use base 16. Since changing the IBM exponent by one means shifting the significand by 4 bits, “normalized” base 16 numbers can have up to 3 leading bits of 0s! Hence, hexadecimal digits mean that up to 3 bits must be dropped from the significand, which leads to surprising problems in the accuracy of floating-point arithmetic, as noted in Section .

Floating-Point Addition

Let’s add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{\text{ten}} \times 10^1 + 1.610_{\text{ten}} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{\text{ten}} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{\text{ten}} \times 10^{-1} = 0.1610_{\text{ten}} \times 10^0 = 0.01610_{\text{ten}} \times 10^1$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{\text{ten}} \times 10^1$. Thus the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really:

$$0.016_{\text{ten}} \times 10^1$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{\text{ten}} \\ + 0.016_{\text{ten}} \\ \hline 10.015_{\text{ten}} \end{array}$$

The sum is $10.015_{\text{ten}} \times 10^1$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{\text{ten}} \times 10^1 = 1.0015_{\text{ten}} \times 10^2$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows

shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

- Step 4. Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{\text{ten}} \times 10^2$$

is rounded to four digits in the significand to

$$1.002_{\text{ten}} \times 10^2$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

Figure 3.16 shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all zero bits in the exponent is reserved and used for the floating-point representation of zero. Also, the pattern of all one bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the Elaboration on page 217). Thus, for single precision, the maximum exponent is 127, and the minimum exponent is −126. The limits for double precision are 1023 and −1022.

EXAMPLE

Decimal Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.16.

ANSWER

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{aligned}
 0.5_{\text{ten}} &= 1/2_{\text{ten}} = 1/2^1_{\text{ten}} \\
 &= 0.1_{\text{two}} = 0.1_{\text{two}} \times 2^0 = 1.000_{\text{two}} \times 2^{-1} \\
 -0.4375_{\text{ten}} &= -7/16_{\text{ten}} = -7/2^4_{\text{ten}} \\
 &= -0.0111_{\text{two}} = -0.0111_{\text{two}} \times 2^0 = -1.110_{\text{two}} \times 2^{-2}
 \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{\text{two}} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned}
 0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\
 &= 1.000_{\text{two}} \times 2^{-4}
 \end{aligned}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{\text{two}} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned}
 1.000_{\text{two}} \times 2^{-4} &= 0.0001000_{\text{two}} = 0.0001_{\text{two}} \\
 &= 1/2^4_{\text{ten}} = 1/16_{\text{ten}} = 0.0625_{\text{ten}}
 \end{aligned}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten} .

Many computers dedicate hardware to run floating-point operations as fast as possible. Figure 3.17 sketches the basic organization of hardware for floating-point addition.

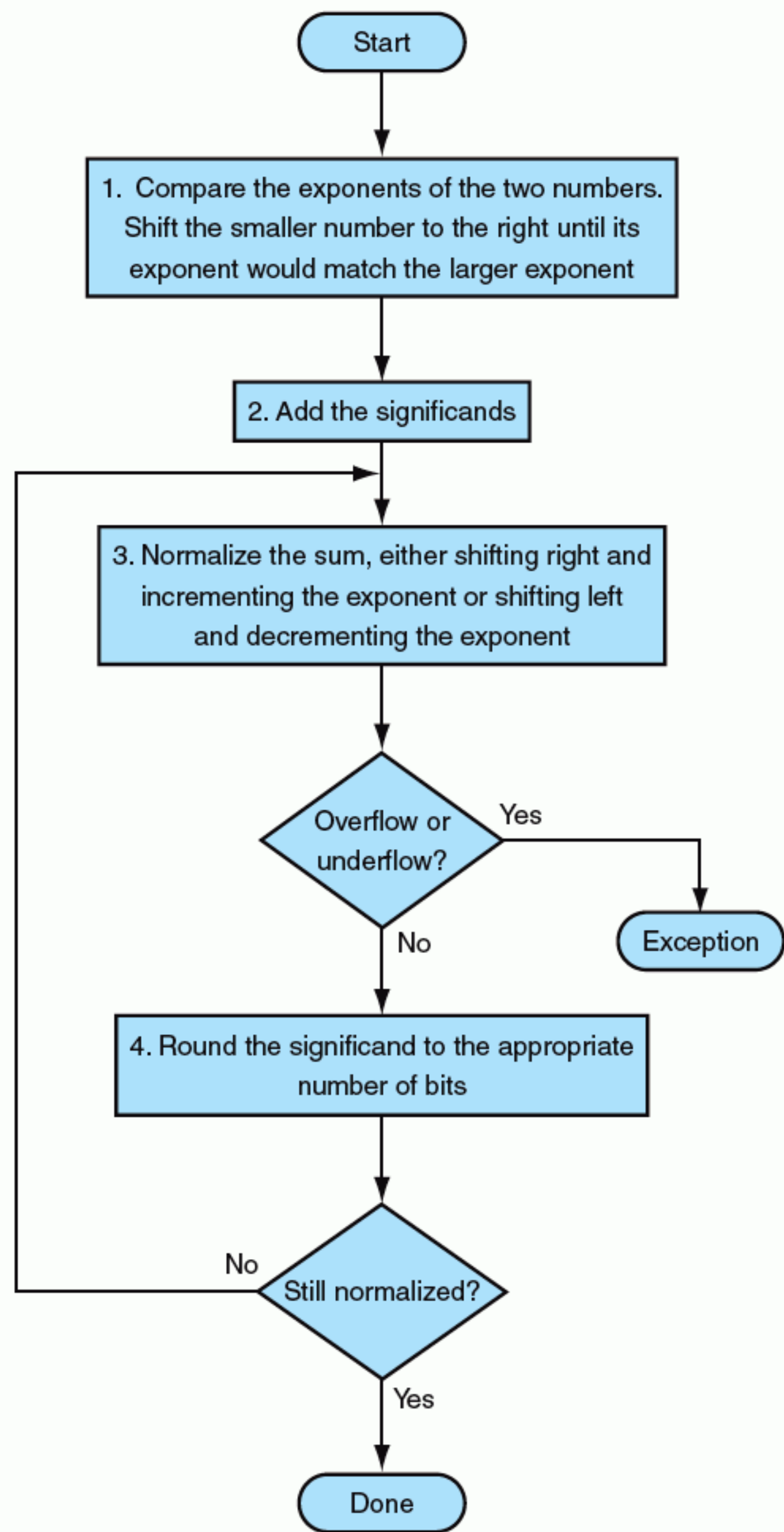


FIGURE 3.16 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

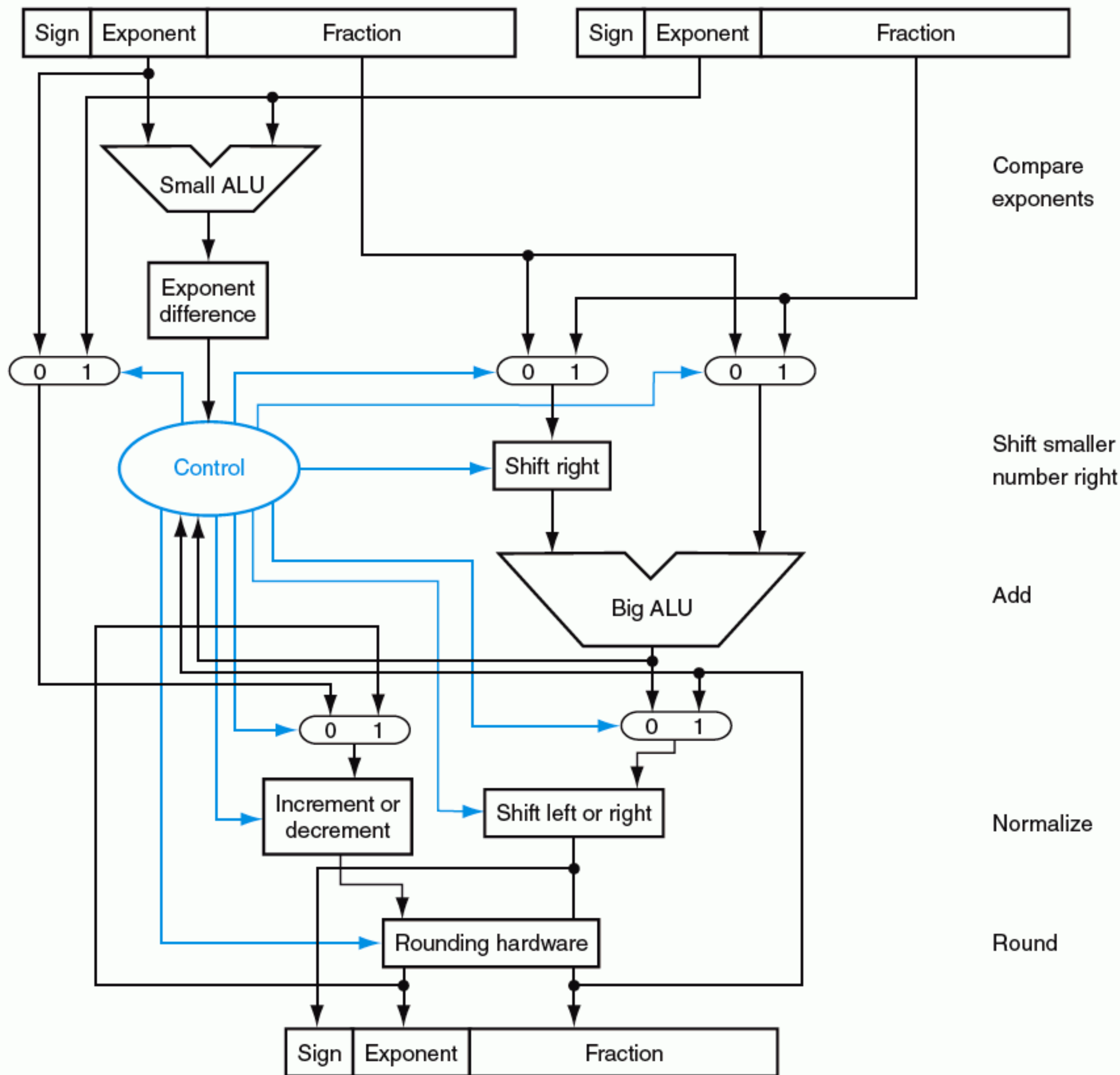


FIGURE 3.17 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.16 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{\text{ten}} \times 10^{10} \times 9.200_{\text{ten}} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: $10 + 127 = 137$, and $-5 + 127 = 122$, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\text{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

$$\begin{array}{r} 1.110_{\text{ten}} \\ \times 9.200_{\text{ten}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{\text{ten}} \end{array}$$

There are three digits to the right of the decimal for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{\text{ten}}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212×10^5 .

Step 3. This product is unnormalized, so we need to normalize it:

$$10.212_{\text{ten}} \times 10^5 = 1.0212_{\text{ten}} \times 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$1.0212_{\text{ten}} \times 10^6$$

is rounded to four digits in the significand to

$$1.021_{\text{ten}} \times 10^6$$

Step 5. The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise it's negative. Hence the product is

$$+1.021_{\text{ten}} \times 10^6$$

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication the sign of the product is determined by the signs of the operands.

Once again, as Figure 3.18 shows, multiplication of binary floating-point numbers is quite similar to the steps we have just completed. We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result. Next is multiplication of significands, followed by an optional normalization step. The size of the exponent is checked for overflow or underflow, and then the product is rounded. If rounding leads to further normalization, we once again check for exponent size. Finally, set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

Decimal Floating-Point Multiplication

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten} , using the steps in Figure 3.18.

EXAMPLE

ANSWER

In binary, the task is multiplying $1.000_{\text{two}} \times 2^{-1}$ by $-1.110_{\text{two}} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$\begin{aligned} (-1 + 127) + (-2 + 127) - 127 &= (-1 - 2) + (127 + 127 - 127) \\ &= -3 + 127 = 124 \end{aligned}$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{\text{two}} \\ \times 1.110_{\text{two}} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{\text{two}} \end{array}$$

The product is $1.110000_{\text{two}} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{\text{two}} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{\text{two}} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{\text{two}} \times 2^{-3} &= -0.001110_{\text{two}} = -0.00111_{\text{two}} \\ &= -7/2^5_{\text{ten}} = -7/32_{\text{ten}} = -0.21875_{\text{ten}} \end{aligned}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten} .

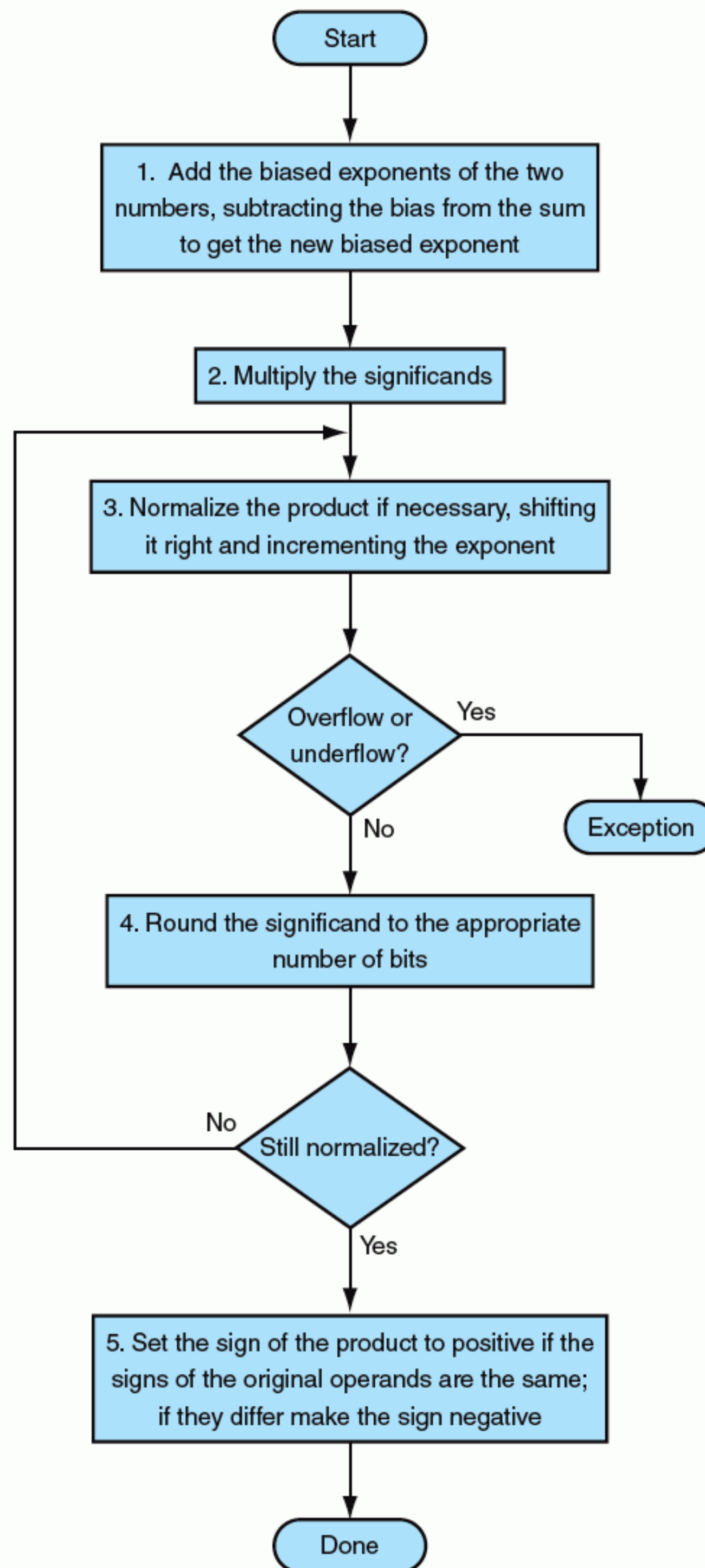


FIGURE 3.18 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Floating-Point Instructions in MIPS

MIPS supports the IEEE 754 single precision and double precision formats with these instructions:

- Floating-point *addition, single* (`add.s`) and *addition, double* (`add.d`)
- Floating-point *subtraction, single* (`sub.s`) and *subtraction, double* (`sub.d`)
- Floating-point *multiplication, single* (`mul.s`) and *multiplication, double* (`mul.d`)
- Floating-point *division, single* (`div.s`) and *division, double* (`div.d`)
- Floating-point *comparison, single* (`c.x.s`) and *comparison, double* (`c.x.d`), where *x* may be *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), or *greater than or equal* (`ge`)
- Floating-point *branch, true* (`bc1t`) and *branch, false* (`bc1f`)

Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.

The MIPS designers decided to add separate floating-point registers—called `$f0`, `$f1`, `$f2`, ...—used either for single precision or double precision. Hence, they included separate loads and stores for floating-point registers: `lwc1` and `swc1`. The base registers for floating-point data transfers remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
lwc1    $f4,x($sp)  # Load 32-bit F.P. number into F4
lwc1    $f6,y($sp)  # Load 32-bit F.P. number into F6
add.s   $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1    $f2,z($sp)  # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.

Figure 3.19 summarizes the floating-point portion of the MIPS architecture revealed in this chapter, with the additions to support floating point shown in color. Similar to Figure 2.25 on page 103 in Chapter 2, we show the encoding of these instructions in Figure 3.20.

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	\$f0, \$f1, \$f2, . . . , \$f31	MIPS floating-point registers are used in pairs for double precision numbers.
2 ³⁰ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6	FP add (single precision)
	FP subtract single	sub.s \$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6	FP sub (single precision)
	FP multiply single	mul.s \$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6	FP multiply (single precision)
	FP divide single	div.s \$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6	FP divide (single precision)
	FP add double	add.d \$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6	FP add (double precision)
	FP subtract double	sub.d \$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6	FP sub (double precision)
	FP multiply double	mul.d \$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6	FP multiply (double precision)
	FP divide double	div.d \$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6	FP divide (double precision)
Data transfer	load word copr. 1	lwc1 \$f1,100(\$s2)	\$f1 = Memory[\$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 \$f1,100(\$s2)	Memory[\$s2 + 100] = \$f1	32-bit data to memory
Conditional branch	branch on FP true	bclt 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s \$f2,\$f4,\$f6
sub.s	R	17	16	6	4	2	1	sub.s \$f2,\$f4,\$f6
mul.s	R	17	16	6	4	2	2	mul.s \$f2,\$f4,\$f6
div.s	R	17	16	6	4	2	3	div.s \$f2,\$f4,\$f6
add.d	R	17	17	6	4	2	0	add.d \$f2,\$f4,\$f6
sub.d	R	17	17	6	4	2	1	sub.d \$f2,\$f4,\$f6
mul.d	R	17	17	6	4	2	2	mul.d \$f2,\$f4,\$f6
div.d	R	17	17	6	4	2	3	div.d \$f2,\$f4,\$f6
lwc1	I	49	20	2	100			lwc1 \$f2,100(\$s4)
swc1	I	57	20	2	100			swc1 \$f2,100(\$s4)
bclt	I	17	8	1	25			bclt 25
bclf	I	17	8	0	25			bclf 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s \$f2,\$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d \$f2,\$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

FIGURE 3.19 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, on page A-49, for more detail.

op(31:26):								
28–26	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
31–29								
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => c = f, rt(16:16) = 1 => c = t), rs(25:21):								
23–21	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
25–24								
0(00)	mfc1		cfcl		mtcl		ctcl	
1(01)	bcl.c							
2(10)	f = single	f = double						
3(11)								

op(31:26) = 010001 (FlPt), (f above: 10000 => f = s, 10001 => f = d), funct(5:0):								
2–0	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
5–3								
0(000)	add.f	sub.f	mul.f	div.f		abs.f	mov.f	neg.f
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.f	cvt.d.f			cvt.w.f			
5(101)								
6(110)	c.f.f	c.un.f	c.eq.f	c.ueq.f	c.olt.f	c.ult.f	c.ole.f	c.ule.f
7(111)	c.sf.f	c.ngle.f	c.seq.f	c.ngl.f	c.lt.f	c.nge.f	c.le.f	c.ngt.f

FIGURE 3.20 MIPS floating-point instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure `lw` is found in row number 4 (`100two` for bits 31–29 of the instruction) and column number 3 (`011two` for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is `100011two`. Underscore means the field is used elsewhere. For example, `FlPt` in row 2 and column 1 (op = `010001two`) is defined in the bottom part of the figure. Hence `sub.f` in row 0 and column 1 of the bottom section means that the funct field (bits 5–0) of the instruction is `000001two` and the op field (bits 31–26) is `010001two`. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (f = s so rs = `10000`) or double precision (f = d so rs = `10001`). Similarly, bit 16 of the instruction determines if the `bcl.c` instruction tests for true (bit 16 = 1 => `bcl.t`) or false (bit 16 = 0 => `bcl.f`). Instructions in color are described in Chapters 2 or 3, with [Appendix A](#) covering all instructions.

One issue that computer designers face in supporting floating-point arithmetic is whether to use the same registers used by the integer instructions or to add a special set for floating point. Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program. The major impact is to create a separate set of data transfer instructions to move data between floating-point registers and memory.

The benefits of separate floating-point registers are having twice as many registers without using up more bits in the instruction format, having twice the register bandwidth by having separate integer and floating-point register sets, and being able to customize registers to floating point; for example, some computers convert all sized operands in registers into a single internal format

Hardware Software Interface

Compiling a Floating-Point C Program into MIPS Assembly Code

Let's convert a temperature in Fahrenheit to Celsius:

```
float f2c (float fahr)
{
    return ((5.0/9.0) * (fahr - 32.0));
}
```

Assume that the floating-point argument `fahr` is passed in `$f12` and the result should go in `$f0`. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

We assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer `$gp`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
    lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
    lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
```

EXAMPLE

ANSWER

They are then divided to get the fraction 5.0/9.0:

```
div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next we load the constant 32.0 and then subtract it from fahr (\$f12):

```
lwc1 $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in \$f0 as the return result, and then return:

```
mul.s $f0, $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr $ra # return
```

Now let's perform floating-point operations on matrices, code commonly found in scientific programs.

EXAMPLE

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of $X = X + Y * Z$. Let's assume X, Y, and Z are all square matrices with 32 elements in each dimension.

```
void mm (double x[][], double y[][], double z[][])
{
    int i, j, k;

    for (i = 0; i != 32; i = i + 1)
        for (j = 0; j != 32; j = j + 1)
            for (k = 0; k != 32; k = k + 1)
                x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

The array starting addresses are parameters, so they are in \$a0, \$a1, and \$a2. Assume that the integer variables are in \$s0, \$s1, and \$s2, respectively. What is the MIPS assembly code for the body of the procedure?

ANSWER

Note that $x[i][j]$ is used in the innermost loop above. Since the loop index is k , the index does not affect $x[i][j]$, so we can avoid loading and storing $x[i][j]$ each iteration. Instead, the compiler loads $x[i][j]$ into a register outside the loop, accumulates the sum of the products of $y[i][k]$ and $z[k][j]$ in that same register, and then stores the sum into $x[i][j]$ upon termination of the innermost loop.

We keep the code simpler by using the assembly language pseudoinstructions `li` (which loads a constant into a register), and `l.d` and `s.d` (which the assembler turns into a pair of data transfer instructions, `lwc1` or `swc1`, to a pair of floating-point registers).

The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables:

```
mm: ...
        li      $t1, 32 # $t1 = 32 (row size/loop end)
        li      $s0, 0  # i = 0; initialize 1st for loop
L1:     li      $s1, 0  # j = 0; restart 2nd for loop
L2:     li      $s2, 0  # k = 0; restart 3rd for loop
```

To calculate the address of $x[i][j]$, we need to know how a 32×32 , two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the i “single-dimensional arrays,” or rows, to get the one we want. Thus we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
sll  $t2, $s0, 5 # $t2 = i * 25 (size of row of x)
```

Now we add the second index to select the j th element of the desired row:

```
addu $t2, $t2, $s1 # $t2 = i * size(row) + j
```

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3:

```
sll  $t2, $t2, 3 # $t2 = byte offset of [i][j]
```

Next we add this sum to the base address of x , giving the address of $x[i][j]$, and then load the double precision number $x[i][j]$ into $\$f4$:

```
addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
l.d  $f4, 0($t2) # $f4 = 8 bytes of x[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number $z[k][j]$.


```

L3:    sll $t0, $s2, 5 # $t0 = k * 25 (size of row of z)
        addu $t0, $t0, $s1 # $t0 = k * size(row) + j
        sll $t0, $t0, 3 # $t0 = byte offset of [k][j]
        addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
        l.d $f16, 0($t0) # $f16 = 8 bytes of z[k][j]

```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number $y[i][k]$.

```

sll     $t2, $s0, 5 # $t0 = i * 25 (size of row of y)
addu    $t0, $t0, $s2 # $t0 = i * size(row) + k
sll     $t0, $t0, 3 # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0 # $t0 = byte address of y[i][k]
l.d     $f18, 0($t0) # $f18 = 8 bytes of y[i][k]

```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of y and z located in registers $\$f18$ and $\$f16$, and then accumulate the sum in $\$f4$.

```

mul.d   $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d   $f4, $f4, $f16 # f4 = x[i][j] + y[i][k] * z[k][j]

```

The final block increments the index k and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in $\$f4$ into $x[i][j]$.

```

addiu   $s2, $s2, 1 # $k = k + 1
bne     $s2, $t1, L3 # if (k != 32) go to L3
s.d     $f4, 0($t2) # x[i][j] = $f4

```

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```

addiu   $s1, $s1, 1 # $j = j + 1
bne     $s1, $t1, L2 # if (j != 32) go to L2
addiu   $s0, $s0, 1 # $i = i + 1
bne     $s0, $t1, L1 # if (i != 32) go to L1
...

```

Elaboration: The array layout discussed in the example, called *row major order*, is used by C and many other programming languages. Fortran instead uses *column major order*, whereby the array is stored column by column.

Only 16 of the 32 MIPS floating-point registers could originally be used for single precision operations: $\$f0, \$f2, \$f4, \dots, \$f30$. Double precision is computed using pairs

of these registers. The odd-numbered floating-point registers were used only to load and store the right half of 64-bit floating-point numbers. MIPS-32 added `l.d` and `s.d` to the instruction set. MIPS-32 also added “paired single” versions of all floating-point instructions, where a single instruction results in two parallel floating-point operations on two 32-bit operands inside 64-bit registers. For example, `add.ps F0, F2, F4` is equivalent to `add.s F0, F2, F4` followed by `add.s F1, F3, F5`.

Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn’t have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence the floating-point unit, including the floating-point registers, were optionally available as a second chip. Such optional accelerator chips are called *coprocessors*, and explain the acronym for floating-point loads in MIPS: `lwc1` means load word to coprocessor 1, the floating-point unit. (Coprocessor 0 deals with virtual memory, described in Chapter 7.) Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and hence the term “coprocessor” joins “accumulator” and “core memory” as quaint terms that date the speaker.

Elaboration: Although there are many ways to throw hardware at floating-point multiply to make it go fast, floating-point division is considerably more challenging to make fast and accurate. Slow divides in early computers led to removal of divides from many algorithms, but parallel computers have inspired rediscovery of divide-intensive algorithms that work better on these computers. Hence, we may need faster divides.

One technique to leverage a fast multiplier is *Newton’s iteration*, where division is recast as finding the zero of a function to find the reciprocal $1/x$, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solves this problem by calculating an extra-precise reciprocal.

Elaboration: Java embraces IEEE 754 by name in its definition of Java floating-point data types and operations. Thus, the code in the first example could have well been generated for a class method that converted Fahrenheit to Celsius.

The second example uses multiple dimensional arrays, which are not explicitly supported in Java. Java allows arrays of arrays, but each array may have its own length, unlike multiple dimensional arrays in C. Like the examples in Chapter 2, a Java version of this second example would require a good deal of checking code for array bounds, including a new length calculation at the end of row. It would also need to check that the object reference is not null.

Accurate Arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can’t really represent. The reason is that an infinite variety of real numbers exists between, say, 0 and 1, but no more than 2^{53} can be represented

guard The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

EXAMPLE

ANSWER

exactly in double precision floating point. The best we can do is get the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

Rounding sounds simple enough, but to round accurately requires the hardware to include extra bits in the calculation. In the preceding examples, we were vague on the number of bits that an intermediate representation can occupy, but clearly if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps 2 extra bits on the right during intermediate additions, called **guard** and **round**, respectively. Let's do a decimal example to illustrate the value of these extra digits.

Rounding with Guard Digits

Add $2.56_{\text{ten}} \times 10^0$ to $2.34_{\text{ten}} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

First we must shift the smaller number to the right to align the exponents, so $2.56_{\text{ten}} \times 10^0$ becomes $0.0256_{\text{ten}} \times 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{\text{ten}} \\ + 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array}$$

Thus the sum is $2.3656_{\text{ten}} \times 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tie-breaker. Rounding the sum up with three significant digits yields $2.37_{\text{ten}} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{\text{ten}} \\ + 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

The answer is $2.36_{\text{ten}} \times 10^2$, off by 1 in the last digit from the sum above.

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of **units in the last place**, or **ulp**. If a number was off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there is no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

units in the last place

(ulp) The number of bits in error in the least significant bits of the significand between the actual number and the number that can be prepresented.

Elaboration: Although the example above really needed just one extra digit, multiply can need two. A binary product may have one leading 0 bit; hence, the normalizing step must shift the product 1 bit left. This shifts the guard digit into the least significant bit of the product, leaving the round bit to help accurately round the product.

There are four rounding modes: always round up (toward $+\infty$), always round down (toward $-\infty$), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The Internal Revenue Service always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one; if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

The goal of the extra rounding bits is to allow the computer to get the same results as if the intermediate results were calculated to infinite precision and then rounded.

To support this goal and rounding to the nearest even, the standard has a third bit in addition to guard and round; it is set whenever there are nonzero bits to the right of the round bit. This **sticky bit** allows the computer to see the difference between $0.50 \dots 00_{\text{ten}}$ and $0.50 \dots 01_{\text{ten}}$ when rounding.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{\text{ten}} \times 10^{-1}$ to $2.34_{\text{ten}} \times 10^2$ in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to $2.345000\dots 00$ and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than $2.345000\dots 00$, we round instead to 2.35.

Summary

The Big Picture below reinforces the stored-program concept from Chapter 2; the meaning of the information cannot be determined just by looking at the bits, for the same bits can represent a variety of objects. This section shows that computer arithmetic is finite and thus can disagree with natural arithmetic. For example, the IEEE 754 standard floating-point representation

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{bias})}$$

is almost always an approximation of the real number. Computer systems must take care to minimize this gap between computer arithmetic and arithmetic in the real world, and programmers at times need to be aware of the implications of this approximation.

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
char	—	lb, sb, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
float	float	lwcl, swcl	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	ld, sd	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

The BIG Picture

Bit patterns have no inherent meaning. They may represent signed integers, unsigned integers, floating-point numbers, instructions, and so on. What is represented depends on the instruction that operates on the bits in the word.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size, hence limited precision; it's possible to calculate a number too big or too small to be represented in a word. Programmers must remember these limits and write programs accordingly.

Hardware Software Interface

In the last chapter we presented the storage classes of the programming language C (see the Hardware Software Interface section on page 142). The table above shows some of the C and Java data types together with the MIPS data transfer instructions and instructions that operate on those types that appear in Chapters 2 and 3. Note that Java omits unsigned integers.

Suppose there was a 16-bit IEEE 754 floating-point format with 5 exponent bits. What would be the likely range of numbers it could represent?

Check Yourself

1. $1.0000\ 0000\ 00 \times 2^0$ to $1.1111\ 1111\ 11 \times 2^{31}$, 0
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \times 2^{15}$, ± 0 , $\pm \infty$, NaN
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \times 2^{15}$, ± 0 , $\pm \infty$, NaN
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ to $\pm 1.1111\ 1111\ 11 \times 2^{14}$, ± 0 , $\pm \infty$, NaN

Elaboration: To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. Hence the full MIPS instruction set has many flavors of compares to support NaNs. (Java does not support unordered compares.)

In an attempt to squeeze every last bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than having a gap between 0 and the smallest normalized number, IEEE allows *denormalized numbers* (also known as *denorms* or *subnormals*). They have the same exponent as zero but a nonzero significand. They allow a number to degrade in significance until it becomes 0, called *gradual underflow*. For example, the smallest positive single precision normalized number is

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{two}} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{\text{two}} \times 2^{-126}, \text{ or } 1.0_{\text{two}} \times 2^{-149}$$

For double precision, the denorm gap goes from 1.0×2^{-1022} to 1.0×2^{-1074} .

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Also, if programmers do not expect denorms, their programs may be surprised.

3.7

Real Stuff: Floating Point in the IA-32

The IA-32 has regular multiply and divide instructions that operate entirely on registers, unlike the reliance on Hi and Lo in MIPS. (In fact, later versions of the MIPS instruction set have added similar instructions.)

The main differences are found in floating-point instructions. The IA-32 floating-point architecture is different from all other computers in the world.

The IA-32 Floating-Point Architecture

The Intel 8087 floating-point coprocessor was announced in 1980. This architecture extended the 8086 with about 60 floating-point instructions.

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the two top elements of the stacks, and stores can pop elements off the stack. Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers on-chip below the top of the stack. Thus, a complete stack instruction set is supplemented by a limited set of register-memory instructions.

This hybrid is still a restricted register-memory model, however, since loads always move data to the top of the stack while incrementing the top-of-stack pointer and stores can only move the top of stack to memory. Intel uses the notation ST to indicate the top of stack, and $ST(i)$ to represent the i th register below the top of stack.

Another novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Unlike the maximum of 64 bits on MIPS, the IA-32 floating-point operands on the stack are 80 bits wide. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. This *double extended precision* is not supported by programming languages, although it has been useful to programmers of mathematical software.

Memory data can be 32-bit (single precision) or 64-bit (double precision) floating-point numbers. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to floating point, and vice versa, for integer loads and stores.

The IA-32 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value
3. Comparison, including instructions to send the result to the integer processor so that it can branch
4. Transcendental instructions, including sine, cosine, log, and exponentiation

Figure 3.21 shows some of the 60 floating-point operations. Note that we get even more combinations when including the operand modes for these operations. Figure 3.22 shows the many options for floating-point add.

Data transfer	Arithmetic	Compare	Transcendental
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

FIGURE 3.21 The floating-point instructions of the IA-32. We use the curly brackets {} to show optional variations of the basic operations: {I} means there is an integer version of the instruction, {P} means this variation will pop one operand off the stack after the operation, and {R} means reverse the order of the operands in this operation. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations in the first column push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the `FSTSW` instruction, either into the `AX` register or into memory, followed by an `SAHF` instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations. Not all combinations suggested by the notation are provided. Hence, `F{I}SUB{R}{P}` operations represents these instructions found in the IA-32: `FSUB`, `FISUB`, `FSUBR`, `FISUBR`, `FSUBP`, `FSUBRP`. For the integer subtract instructions, there is no pop (`FISUBP`) or reverse pop (`FISUBRP`).

Instruction	Operands	Comment
FADD		Both operands in stack; result replaces top of stack.
FADD	ST(i)	One source operand is <i>i</i> th register below the top of stack; result replaces the top of stack.
FADD	ST(i), ST	One source operand is the top of stack; result replaces <i>i</i> th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory; result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory; result replaces the top of stack.

FIGURE 3.22 The variations of operands for floating-point add in the IA-32.

The floating-point instructions are encoded using the `ESC` opcode of the 8086 and the postbyte address specifier (see Figure 2.46 on page 143). The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit floating point or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Floating-point performance of the IA-32 family has traditionally lagged far behind other computers. Whether it is simply a lack of attention by Intel engineers or a flaw with its architecture is hard to know. We can say that many new architectures have been announced since 1980, and none have followed in Intel's footsteps. In addition, Intel created a more traditional floating-point architecture as part of SSE2.

The Intel Streaming SIMD Extension 2 (SSE2) Floating-Point Architecture

Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture, including double precision floating-point registers and operations. It includes eight registers that can be used for floating-point operands, giving the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE2 registers as floating-point registers like those found in other computers. AMD expanded the number to 16 as part of AMD64, which Intel relabeled EM64T for its use.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles or two doubles. This new architecture can more than double performance over the stack architecture.

Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.

Bertrand Russell, *Recent Words on the Principles of Mathematics*, 1901

3.8

Fallacies and Pitfalls

Arithmetic fallacies and pitfalls generally stem from the difference between the limited precision of computer arithmetic and the unlimited precision of natural arithmetic.

Fallacy: Floating-point addition is associative; that is, $x + (y + z) = (x + y) + z$.

Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number.

For example, suppose $x = -1.5_{\text{ten}} \times 10^{38}$, $y = 1.5_{\text{ten}} \times 10^{38}$, and $z = 1.0$, and that these are all single precision numbers. Then

$$\begin{aligned} x + (y + z) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\ &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) = 0.0 \\ (x + y) + z &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\ &= (0.0_{\text{ten}}) + 1.0 \\ &= 1.0 \end{aligned}$$

Therefore, $x + (y + z) \neq (x + y) + z$.

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{\text{ten}} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{\text{ten}} \times 10^{38} + 1.0$ is still $1.5_{\text{ten}} \times 10^{38}$. That is why the sum of x , y , and z is 0.0 or 1.0, depending on the order of the floating-point additions, and hence floating-point add is *not* associative.

Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.

Recall that a binary number x , where x_i means the i th bit, represents the number

$$\dots + (x_3 \times 2^3) + (x_2 \times 2^2) + (x_1 \times 2^1) + (x_0 \times 2^0)$$


Shifting the bits of x right by n bits would seem to be the same as dividing by 2^n . And this *is* true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide -5_{ten} by 4_{ten} ; the quotient should be -1_{ten} . The two's complement representation of -5_{ten} is

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{\text{two}}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $1,073,741,822_{\text{ten}}$ instead of -1_{ten} .

A solution would be to have an arithmetic right shift (see  **In More Depth:** Booth's Algorithm) that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

The result is -2_{ten} instead of -1_{ten} ; close, but no cigar.

The PowerPC, however, does have a fast shift instruction (*shift right algebraic*) that in conjunction with a special add (add with carry) gives the same answer as dividing by a power of 2.

Pitfall: The MIPS instruction add immediate unsigned addiu sign-extends its 16-bit immediate field.

Despite its name, add immediate unsigned (addiu) is used to add constants to signed integers when we don't care about overflow. MIPS has no subtract immediate instruction and negative numbers need sign extension, so the MIPS architects decided to sign-extend the immediate field.

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 3.23). The following is the inside story behind the headlines.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing -2 , -1 , 0 , $+1$, or $+2$. The guess is multiplied by the divisor and



FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the *New York Times*, *San Jose Mercury News*, *San Francisco Chronicle*, and *Infoworld*. The Pentium floating-point divide bug even made the “Top 10 List” of the *David Letterman Late Show* on television. Intel eventually took a \$300 million write-off to replace the buggy chips.

subtracted from the remainder to generate a new remainder. Like nonrestoring division (see Exercise 3.29), if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently there were five elements of the table from the 80486 that Intel thought could never be accessed, and they optimized the PLA to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11 bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

The following is a time line of the Pentium bug morality play:

- *July 1994:* Intel discovers the bug in the Pentium. The actual cost to fix the bug was several hundred thousand dollars. Following normal bug fix procedures, it will take months to make the change, reverify, and put the corrected chip into production. Intel planned to put good chips into production in January 1995, estimating that 3 to 5 million Pentiums would be produced with the bug.
- *September 1994:* A math professor at Lynchburg College in Virginia, Thomas Nicely, discovers the bug. After calling Intel technical support and getting no official reaction, he posts his discovery on the Internet. It quickly gained a following, and some pointed out that even small errors become big when multiplying by big numbers: the fraction of people with a rare disease times the population of Europe, for example, might lead to the wrong estimate of the number of sick people.
- *November 7, 1994:* *Electronic Engineering Times* puts the story on its front page, which is soon picked up by other newspapers.
- *November 22, 1994:* Intel issues a press release, calling it a “glitch.” The Pentium “can make errors in the ninth digit. . . . Even most engineers and financial analysts require accuracy only to the fourth or fifth decimal point. Spreadsheet and word processor users need not worry. . . . There are maybe several dozen people that this would affect. So far, we’ve only heard from one. . . . [Only] theoretical mathematicians (with Pentium computers purchased before the summer) should be concerned.” What irked many was that customers were told to describe their application to Intel, and then *Intel* would decide whether or not their application merited a new Pentium without the divide bug.
- *December 5, 1994:* Intel claims the flaw happens once in 27,000 years for the typical spreadsheet user. Intel assumes a user does 1000 divides per day and multiplies the error rate assuming floating-point numbers are random, which is one in 9 billion, and then gets 9 million days, or 27,000 years.

Things begin to calm down, despite Intel neglecting to explain why a typical customer would access floating-point numbers randomly.

- *December 12, 1994:* IBM Research Division disputes Intel's calculation of the rate of errors (you can access this article by visiting www.mkp.com/books_catalog/cod/links.htm). IBM claims that common spreadsheet programs, recalculating for 15 minutes a day, could produce Pentium-related errors as often as once every 24 days. IBM assumes 5000 divides per second, for 15 minutes, yielding 4.2 million divides per day, and does not assume random distribution of numbers, instead calculating the chances as one in 100 million. As a result, IBM immediately stops shipment of all IBM personal computers based on the Pentium. Things heat up again for Intel.
- *December 21, 1994:* Intel releases the following, signed by Intel's president, chief executive officer, chief operating officer, and chairman of the board: "We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer." Analysts estimate that this recall cost Intel \$500 million, and Intel employees did not get a Christmas bonus that year.

This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

In April 1997 another floating-point bug was revealed in the Pentium Pro and Pentium II microprocessors. When the floating-point-to-integer store instructions (`fist`, `fistp`) encounter a negative floating-point number that is too large to fit in a 16- or 32-bit word after being converted to integer, they set the wrong bit in the FPO status word (precision exception instead of invalid operation exception). To Intel's credit, this time they publicly acknowledged the bug and offered a software patch to get around it—quite a different reaction from what they did in 1994.

3.9**Concluding Remarks**

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called “overflow” or “underflow,” may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. Chapter 5 discusses exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation are part of the inspiration for the field of numerical analysis.

Over the years, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two’s complement binary integer arithmetic and IEEE 754 binary floating-point arithmetic are found in the vast majority of computers sold today. For example, every desktop computer sold since this book was first printed follows these conventions.

A side effect of the stored-program computer is that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, instruction, and so on. It is the instruction that operates on the word that determines its meaning.

With the explanation of computer arithmetic in this chapter comes a description of much more of the MIPS instruction set. One point of confusion is the instructions covered in these chapters versus instructions executed by MIPS chips versus the instructions accepted by MIPS assemblers. The next two figures try to make this clear.

Figure 3.24 lists the MIPS instructions covered in this chapter and Chapter 2. We call the set of instructions on the left-hand side of the figure the *MIPS core*. The instructions on the right we call the *MIPS arithmetic core*. On the left of Figure 3.25 are the instructions the MIPS processor executes that are not found in Figure 3.24. We call the full set of hardware instructions *MIPS-32*. On the right of Figure 3.25 are the instructions accepted by the assembler that are not part of MIPS-32. We call this set of instructions *Pseudo MIPS*.

Instruction subset	Integer	Fl. pt.
MIPS core	95%	57%
MIPS arithmetic core	0%	41%
Remaining MIPS-32	5%	2%

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mghi	R
subtract unsigned	subu	R	move from Lo	mflo	R
and	and	R	move from system control (EPC)	mfc0	R
and immediate	andi	I	floating-point add single	add.s	R
or	or	R	floating-point add double	add.d	R
or immediate	ori	I	floating-point subtract single	sub.s	R
nor	nor	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdcl	I
store byte	sb	I	branch on floating-point true	bclt	I
branch on equal	beq	I	branch on floating-point false	bclf	I
branch on not equal	bne	I	floating-point compare single	c.x.s	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J	floating-point compare double	c.x.d	R
jump register	jr	R	(x = eq, neq, lt, le, gt, ge)		
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURE 3.24 The MIPS instruction set covered so far. This book concentrates on the instructions in the left column.

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or ($rs \oplus rt$)	xor	R	move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
shift right arithmetic	sra	R	not ($\neg rs$)	not	rd,rs
shift left logical variable	sllv	R	negate (<i>signed or unsigned</i>)	neg _s	rd,rs
shift right logical variable	srlv	R	rotate left	rol	rd,rs,rt
shift right arithmetic variable	srav	R	rotate right	ror	rd,rs,rt
move to Hi	mthi	R	multiply and don't check oflw (<i>signed or uns.</i>)	mul _s	rd,rs,rt
move to Lo	mtlo	R	multiply and check oflw (<i>signed or uns.</i>)	mulos	rd,rs,rt
load halfword	lh	I	divide and check overflow	div	rd,rs,rt
load byte	lb	I	divide and don't check overflow	divu	rd,rs,rt
load word left (<i>unaligned</i>)	lwl	I	remainder (<i>signed or unsigned</i>)	rem _s	rd,rs,rt
load word right (<i>unaligned</i>)	lwr	I	load immediate	li	rd,imm
store word left (<i>unaligned</i>)	swl	I	load address	la	rd,addr
store word right (<i>unaligned</i>)	swr	I	load double	ld	rd,addr
load linked (<i>atomic update</i>)	ll	I	store double	sd	rd,addr
store cond. (<i>atomic update</i>)	sc	I	unaligned load word	ulw	rd,addr
move if zero	movz	R			
move if not zero	movn	R	unaligned store word	usw	rd,addr
multiply and add (S or <i>uns.</i>)	madd _s	R			
multiply and subtract (S or <i>uns.</i>)	msub _s	I	unaligned load halfword (<i>signed or uns.</i>)	ulhs	rd,addr
branch on \geq zero and link	bgezal	I	unaligned store halfword	ush	rd,addr
branch on $<$ zero and link	bltzal	I	branch	b	Label
jump and link register	jalr	R	branch on equal zero	beqz	rs,L
branch compare to zero	bxz	I	branch on compare (<i>signed or unsigned</i>)	bx _s	rs,rt,L
branch compare to zero likely	bxzl	I	($x = lt, le, gt, ge$)		
($x = lt, le, gt, ge$)			set equal	seq	rd,rs,rt
branch compare reg likely	bxl	I	set not equal	sne	rd,rs,rt
trap if compare reg	tx	R	set on compare (<i>signed or unsigned</i>)	sx _s	rd,rs,rt
trap if compare immediate	txi	I	($x = lt, le, gt, ge$)		
($x = eq, neq, lt, le, gt, ge$)			load to floating point (<i>s or d</i>)	l. _f	rd,addr
return from exception	rfe	R	store from floating point (<i>s or d</i>)	s. _f	rd,addr
system call	syscall	I			
break (<i>cause exception</i>)	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move (<i>s or d</i>)	mov. _f	R			
FP move if zero (<i>s or d</i>)	movz. _f	R			
FP move if not zero (<i>s or d</i>)	movn. _f	R			
FP square root (<i>s or d</i>)	sqr. _f	R			
FP absolute value (<i>s or d</i>)	abs. _f	R			
FP negate (<i>s or d</i>)	neg. _f	R			
FP convert (<i>w, s, or d</i>)	cvt. _{f.f}	R			
FP compare un (<i>s or d</i>)	c.xn. _f	R			

FIGURE 3.25 Remaining MIPS-32 and “Pseudo MIPS” instruction sets. *f* means single (s) or double precision (d) floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd._f/msub._f), ceiling (ceil._f), truncate (trunc._f), round (round._f), and reciprocal (recip._f).

Figure 3.26 gives the popularity of the MIPS instructions for SPEC2000 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 1% of the instructions executed. The following table summarizes that information.

Note that although programmers and compiler writers may use MIPS-32 to have a richer menu of options, MIPS core instructions dominate integer SPEC2000 execution, and the integer core plus arithmetic core dominate SPEC2000 floating point.


For the rest of the book, we concentrate on the MIPS core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. As we can see, the MIPS core includes the most popular MIPS instructions, and be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious computers.

Core MIPS	Name	Integer	Fl. pt.	Arithmetic core + MIPS-32	Name	Integer	Fl. pt.
add	add	0%	0%	FP add double	add.d	0%	8%
add immediate	addi	0%	0%	FP subtract double	sub.d	0%	3%
add unsigned	addu	7%	21%	FP multiply double	mul.d	0%	8%
add immediate unsigned	addiu	12%	2%	FP divide double	div.d	0%	0%
subtract unsigned	subu	3%	2%	load word to FP double	l.d	0%	15%
and	and	1%	0%	store word to FP double	s.d	0%	7%
and immediate	andi	3%	0%	shift right arithmetic	sra	1%	0%
or	or	7%	2%	load half	lhu	1%	0%
or immediate	ori	2%	0%	branch less than zero	bltz	1%	0%
nor	nor	3%	1%	branch greater or equal zero	bgez	1%	0%
shift left logical	sll	1%	1%	branch less or equal zero	blez	0%	1%
shift right logical	srl	0%	0%	multiply	mul	0%	1%
load upper immediate	lui	2%	5%				
load word	lw	24%	15%				
store word	sw	9%	2%				
load byte	lbu	1%	0%				
store byte	sb	1%	0%				
branch on equal (zero)	beq	6%	2%				
branch on not equal (zero)	bne	5%	1%				
jump and link	jal	1%	0%				
jump register	jr	1%	0%				
set less than	slt	2%	0%				
set less than immediate	slti	1%	0%				
set less than unsigned	sltu	1%	0%				
set less than imm. uns.	sltiu	1%	0%				

FIGURE 3.26 The frequency of the MIPS instructions for SPEC2000 integer and floating point. All instructions that accounted for at least 1% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here. This data is from Chapter 2 of *Computer Architecture: A Quantitative Approach*, third edition.



Historical Perspective and Further Reading

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, plus the rationale for the 80-bit stack architecture for floating point in the IA-32. See  [Section 3.10](#).

Gresham's Law (“Bad money drives out Good”) for computers would say, “The Fast drives out the Slow even if the Fast is wrong.”

W. Kahan, 1992



Exercises

Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.

Winston Churchill, address at Harrow School, 1941, *Abroad*, 1869

- 3.1** [3] <§3.2> Convert 4096_{ten} into a 32-bit two’s complement binary number.
- 3.2** [3] <§3.2> Convert -2047_{ten} into a 32-bit two’s complement binary number.
- 3.3** [5] <§3.2> Convert $-2,000,000_{\text{ten}}$ into a 32-bit two’s complement binary number.
- 3.4** [5] <§3.2> What decimal number does this two’s complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0000\ 0110_{\text{two}}$?
- 3.5** [5] <§3.2> What decimal number does this two’s complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_{\text{two}}$?
- 3.6** [5] <§3.2> What decimal number does this two’s complement binary number represent: $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_{\text{two}}$?
- 3.7** [10] <§3.2> Find the shortest sequence of MIPS instructions to determine the absolute value of a two’s complement integer. Convert this instruction (accepted by the MIPS assembler):

```
abs    $t2,$t3
```

This instruction means that register $\$t2$ has a copy of register $\$t3$ if register $\$t3$ is positive, and the two’s complement of register $\$t3$ if $\$t3$ is negative. (Hint: It can be done with three instructions.)

- 3.8** [10] <§3.2>  [For More Practice](#): Number Representations

3.9 [10] <§3.2> If A is a 32-bit address, typically an instruction sequence such as

```
lui $t0, A_upper  
ori $t0, $t0, A_lower  
lw $s0, 0($t0)
```

can be used to load the word at A into a register (in this case, \$s0). Consider the following alternative, which is more efficient:

```
lui $t0, A_upper_adjusted  
lw $s0, A_lower($t0)
```

Describe how A_upper is adjusted to allow this simpler code to work. (Hint: A_upper needs to be adjusted because A_lower will be sign-extended.)


3.10 [10] <§3.3> Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers, say, registers \$t3 and \$t4. Place a 0 or 1 in register \$t2 if the carry out is 0 or 1, respectively. (Hint: It can be done in two instructions.)

3.11 [15] <§3.3>  **For More Practice:** Writing MIPS Code to Perform Arithmetic

3.12 [15] <§3.3> Suppose that all of the conditional branch instructions except beq and bne were removed from the MIPS instruction set along with slt and all of its variants (slti, sltu, sltui). Show how to perform

```
slt $t0, $s0, $s1
```

using the modified instruction set in which slt is not available. (Hint: It requires more than two instructions.)

3.13 [10] <§3.3> Draw the gates for the Sum bit of an adder, given the equation on  page B-28.

3.14 [5] <§3.4>  **For More Practice:** Writing MIPS Code to Perform Arithmetic

3.15 [20] <§3.4>  **For More Practice:** Writing MIPS Code to Perform Arithmetic

3.16 [2 weeks] <§3.4>  **For More Practice:** Simulating MIPS Machines

3.17 [1 week] <§3.4>  **For More Practice:** Simulating MIPS Machines

3.18 [5] <§3.4>  **For More Practice:** Carry Lookahead Adders

3.19 [15] <§3.4>  **For More Practice:** Carry Lookahead Adders

3.20 [10] <§3.4>  **For More Practice:** Relative Performance of Adders


3.21 [15] <§3.4>  **For More Practice:** Relative Performance of Adders

3.22 [15] <§3.4>  **For More Practice:** Relative Performance of Adders

3.23 [30] <§3.4>  **In More Depth:** Booth's Algorithm

3.24 [15][30] <§3.4>  **For More Practice:** Special MIPS Registers

3.25 [10] <§§3.5, 3.4>  **In More Depth:** The Power PC's Multiply-Add Instruction

3.26 [20] <§3.5>  **In More Depth:** The Power PC's Multiply-Add Instruction

3.27 <§§3.3, 3.4, 3.5> With $x = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 1011_{\text{two}}$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}}$ representing two's complement signed integers, perform, showing all work:

- a. $x + y$
- b. $x - y$
- c. $x * y$
- d. x/y

3.28 [20] <§§3.3, 3.4, 3.5> Perform the same operations as Exercise 3.27, but with $x = 1111\ 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1101\ 0111_{\text{two}}$.

3.29 [30] <§3.5> The division algorithm in Figure 3.11 on page 185 is called *restoring division*, since each time the result of subtracting the divisor from the dividend is negative you must add the divisor back into the dividend to restore the original value. Recall that shift left is the same as multiplying by 2. Let's look at the value of the left half of the Remainder again, starting with step 3b of the divide algorithm and then going to step 2:

$$(\text{Remainder} + \text{Divisor}) \times 2 - \text{Divisor}$$

This value is created from restoring the Remainder by adding the Divisor, shifting the sum left, and then subtracting the Divisor. Simplifying the result we get

$$\text{Remainder} \times 2 + \text{Divisor} \times 2 - \text{Divisor} = \text{Remainder} \times 2 + \text{Divisor}$$


Based on this observation, write a *nonrestoring division* algorithm using the notation of Figure 3.11 that does not add the Divisor to the Remainder in step 3b. Show that your algorithm works by dividing $0000\ 1011_{\text{two}}$ by 0011_{two} .

3.30 [15] <§§3.2, 3.6> The Big Picture on page 216 mentions that bits have no inherent meaning. Given the bit pattern:

1010 1101 0001 0000 0000 0000 0010

what does it represent, assuming that it is

- a two's complement integer?
- an unsigned integer?
- a single precision floating-point number?
- a MIPS instruction?

You may find Figures 3.20 (page 208) and  [A.10.2](#) (page A-50) useful.

3.31 <§§3.2, 3.6> This exercise is similar to Exercise 3.30, but this time use the bit pattern

0010 0100 1001 0010 0100 1001 0010 0100

3.32 [10] [10] <§3.6>  **For More Practice:** Floating Point Number Representations

3.33 [10] <§3.6>  **For More Practice:** Floating Point Number Representations

3.34 [10] <§3.6>  **For More Practice:** Writing MIPS Code to Perform FP Arithmetic

3.35 [5] <§3.6> Add $2.85_{\text{ten}} \times 10^3$ to $9.84_{\text{ten}} \times 10^4$, assuming that you have only three significant digits, first with guard and round digits and then without them.

3.36 [5] <§3.6> This exercise is similar to Exercise 3.35, but this time use the numbers $3.63_{\text{ten}} \times 10^4$ and $6.87_{\text{ten}} \times 10^3$.

3.37 [5] <§3.6> Show the IEEE 754 binary representation for the floating-point number 20_{ten} in single and double precision.

3.38 [5] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 20.5_{ten} .

3.39 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 0.1_{ten} .

3.40 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with the decimal fraction $-5/6$.

3.41 [10] <§3.6> Suppose we introduce a new instruction that adds three floating-point numbers. Assuming we add them together with a triple adder, with guard, round, and sticky bits, are we guaranteed results within 1 ulp of the results using two distinct add instructions?

3.42 [15] <§3.6> With $x = 0100\ 0110\ 1101\ 1000\ 0000\ 0000\ 0000\ 0000_{\text{two}}$ and $y = 1011\ 1110\ 1110\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

- a. $x + y$
- b. $x * y$

3.43 [15] <§3.6> With $x = 0101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{\text{two}}$, $y = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_{\text{two}}$, and $z = 1101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{\text{two}}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

- a. $x + y$
- b. (result of a) + z
- c. Why is this result counterintuitive?

3.44 [20] <§§3.6, 3.7> The IEEE 754 floating-point standard specifies 64-bit double precision with a 53-bit significand (including the implied 1) and an 11-bit exponent. IA-32 offers an extended precision option with a 64-bit significand and a 16-bit exponent.

- a. Assuming extended precision is similar to single and double precision, what is the bias in the exponent?
- b. What is the range of numbers that can be represented by the extended precision option?
- c. How much greater is this accuracy compared to double precision?

3.45 [5] <§§3.6, 3.7> The internal representation of floating point numbers in IA-32 is 80 bits wide. This contains a 16 bit exponent. However it also advertises a 64 bit significand. How is this possible?

3.46 [10] <§3.7> While the IA-32 allows 80-bit floating-point numbers internally, only 64-bit floating-point numbers can be loaded or stored. Starting with only 64-bit numbers, how many operations are required before the full range of the 80-bit exponents are used? Give an example.

3.47 [25] <§3.8>  **For More Practice:** Floating Point on Algorithms

3.48 [30] <§3.8>  **For More Practice:** Floating Point Rounding Modes

3.49 [30] <§3.8>  **For More Practice:** Denormalized Numbers

3.50 [10] <§3.9>  **For More Practice:** Evaluating Instruction Frequencies

3.51 [10] <§3.9>  **For More Practice:** Evaluating Instruction Frequencies

3.52 [10] <§3.9>  **For More Practice:** Evaluating Instruction Frequencies

3.53 [10] <§3.9>  **For More Practice:** Evaluating Instruction Frequencies

3.54 [15] <§3.9>  **For More Practice:** Evaluating Performance

3.55 [15] <§3.9>  **For More Practice:** Evaluating Performance

Answers To Check Yourself

§3.2, page 168: 3, since each character in a Java string takes 16 bits plus one word for length.

§3.3, page 174: 2.

§3.6, page 217: 3.

Computers in the Real World

Reconstructing the Ancient World

Problem: Analyzing and understanding archeological sites is challenging. Can we find ways to use computers to help researchers to explore archeological sites and artifacts discovered at those sites?

Solution: Archeology is undergoing a revolution with the use of digital tools for mapping out ancient sites, reconstructing damaged artifacts, and recreating ancient sites in three dimensions. Among the important new techniques being used to analyze and recreate sites are the following:

- The use of geographical information systems (GIS) to help accurately measure sites. GIS uses global positioning systems (GPS) to accurately pinpoint locations, allowing fast and precise measurements of a site.
- Laser range finding to obtain accurate measurements of the two- and three-dimensional structure of objects. Laser range-finding is even being used with low-flying aircraft to obtain height measurements.



A digital photograph taken from a virtual reality model of the new temple at Chavín de Huántar.

- Digital photography to obtain accurate images of sites as well as individual objects.
- Virtual reality and three-dimensional visualization systems can use digital photographic data and accurate geospatial information to create realistic versions of archeological sites, allowing archeolo-

gists to gain new insights, as well as to share their work with other researchers and the general public.

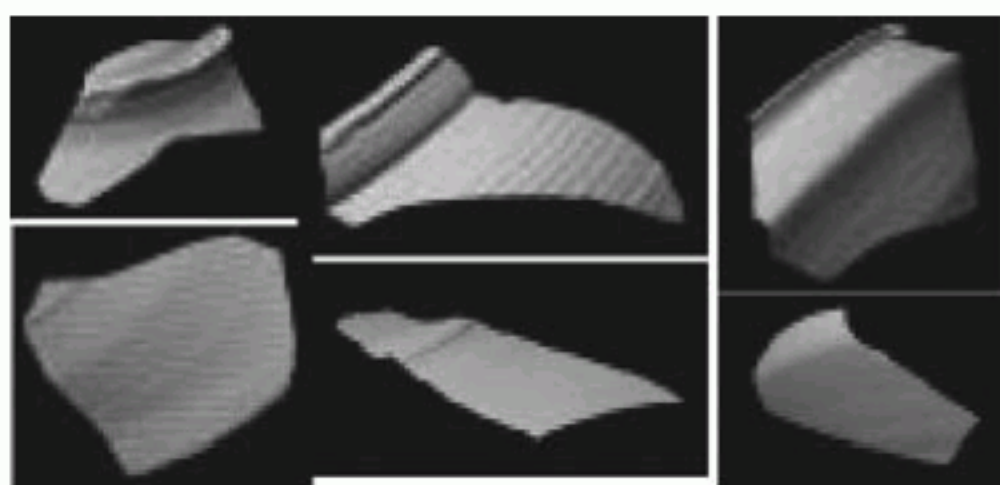
These techniques have been used to explore and create an interactive virtual reality model of an archeological site called Chavín de Huántar, which is in the Peruvian highlands. The image on the top of the previous page is a digital still photograph taken from the virtual reality model. Chavín de Huántar was occupied from about 1000 BCE and predates the classical Incan civilization by more than 1000 years. Highly detailed photos together with measurements of over 25,000 points allows a reconstruction of an accurate virtual model. The image on the top of the previous page is from the new temple at Chavín, which played

a key role in the establishment of formalized religious authority in the New World.

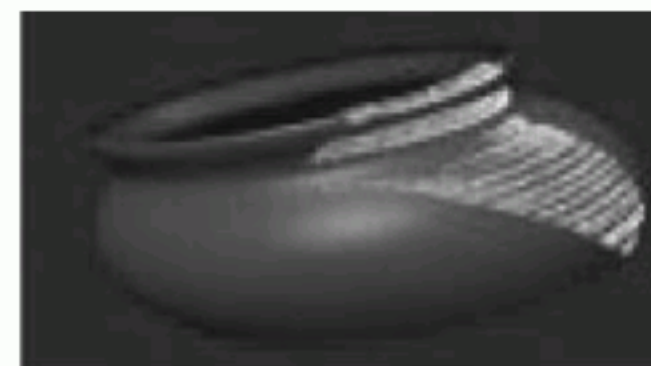
Three-dimensional modeling and reconstruction have also been used in the reconstruction of artifacts from fragments. The images on the left below are fragments of pottery found at Petra, the famous archeological site in Jordan. On the right is a computer reconstruction of the original vessel, highlighting the position of one of the fragments.

To learn more see these references on the  library

- Reconstructing objects from fragments at the SHAPE Laboratory at Brown University
- The Chavín de Huántar exploration (includes virtual reality tour of the site)



Images of pottery fragments found at Petra, Jordan




A computer reconstruction from the fragments in the previous photo

4

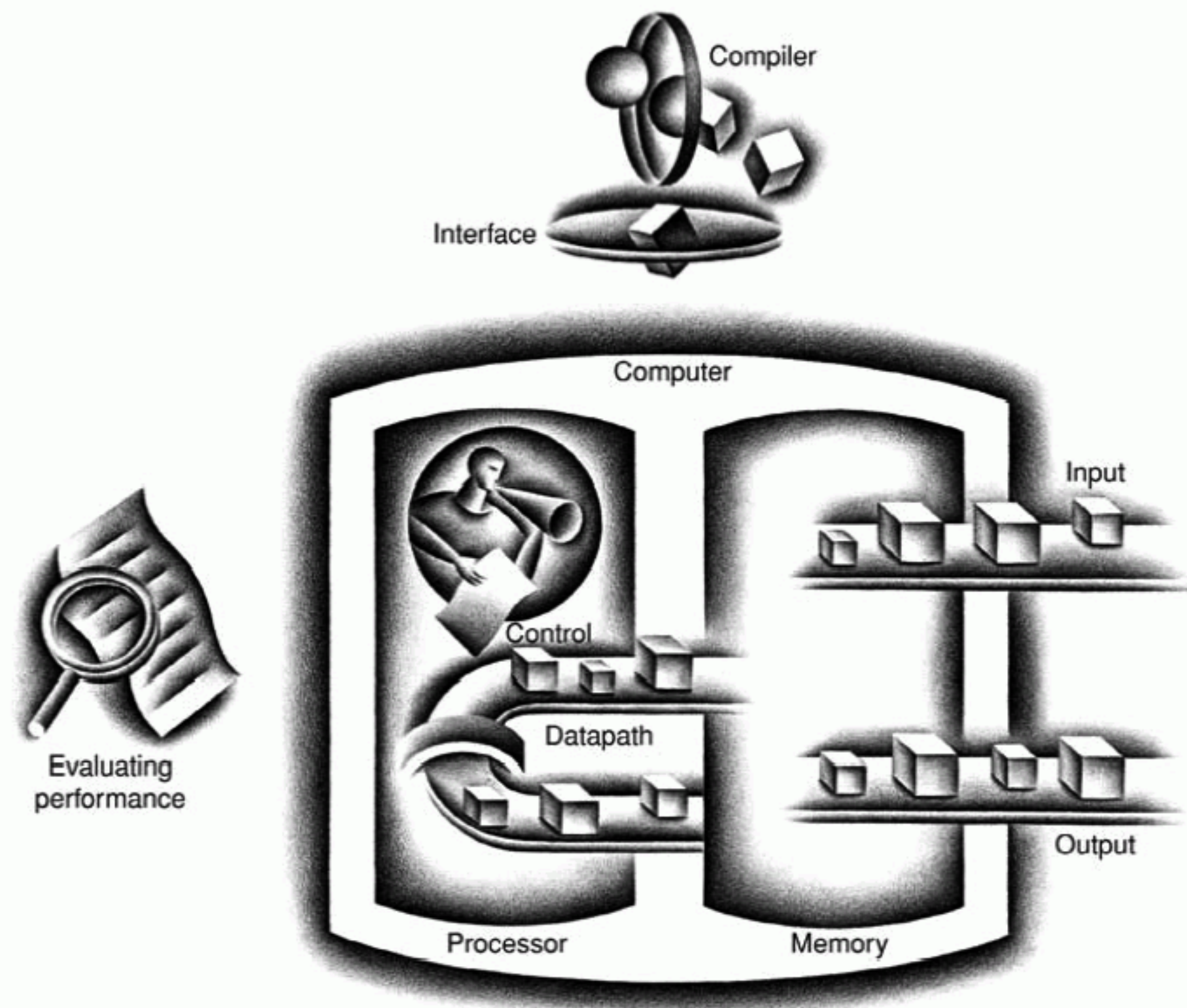
Assessing and Understanding Performance

Time discovers truth.

Seneca

4.1	Introduction	240
4.2	CPU Performance and Its Factors	246
4.3	Evaluating Performance	254
4.4	Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors	259
4.5	Fallacies and Pitfalls	266
4.6	Concluding Remarks	270
 4.7	Historical Perspective and Further Reading	272
4.8	Exercises	272

The Five Classic Components of a Computer



4.1 Introduction

This chapter discusses how to measure, report, and summarize performance and describes the major factors that determine the performance of a computer. A primary reason for examining performance is that hardware performance is often key to the effectiveness of an entire system of hardware and software. Much of the material in this chapter, especially this section and the next, will be critical to understanding the next three chapters. The rest of the chapter provides important insights and principles that anyone seeking to evaluate the performance of a computer system should know. The material in Sections 4.3–4.5, however, is not necessary for the immediately following chapters and can be returned to later.

Assessing the performance of such a system can be quite challenging. The scale and intricacy of modern software systems, together with the wide range of performance improvement techniques employed by hardware designers, have made performance assessment much more difficult. It is simply impossible to sit down with an instruction set manual and a significant software system and determine, simply by analysis, how fast the software will run on the computer. In fact, for different types of applications, different performance metrics may be appropriate, and different aspects of a computer system may be the most significant in determining overall performance.

Of course, in trying to choose among different computers, performance is almost always an important attribute. Accurately measuring and comparing different computers is critical to purchasers, and therefore to designers. The people selling computers know this as well. Often, salespeople would like you to see their computer in the best possible light, whether or not this light accurately reflects the needs of the purchaser's application. In some cases, claims are made about computers that don't provide useful insight for any real applications. Hence, understanding how best to measure performance and the limitations of performance measurements is important in selecting a computer.

Our interest in performance, however, goes beyond issues of assessing performance only from the outside of a computer. To understand why a piece of software performs as it does, why one instruction set can be implemented to perform better than another, or how some hardware feature affects performance, we need to understand what determines the performance of a computer. For example, to improve the performance of a software system, we may need to understand what factors in the hardware contribute to the overall system performance and the relative importance of these factors. These factors may include how well the program uses the instructions of the computer, how well the underlying hardware

implements the instructions, and how well the memory and I/O systems perform. Understanding how to determine the performance impact of these factors is crucial to understanding the motivation behind the design of particular aspects of the computer, as we will see in the chapters that follow.

The rest of this section describes different ways in which performance can be determined. In Section 4.2, we describe the metrics for measuring performance from the viewpoint of both a computer user and a designer. We also look at how these metrics are related and present the classical processor performance equation, which we will use throughout the text. Sections 4.3 and 4.4 describe how best to choose benchmarks to evaluate computers and how to accurately summarize the performance of a group of programs. Section 4.4 also describes one set of commonly used CPU benchmarks and examines measurements for a variety of Intel processors using those benchmarks. Finally, in Section 4.5, we’ll examine some of the many pitfalls that have trapped designers and those who analyze and report performance.

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. Figure 4.1 shows some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed is the Concorde, the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

Let’s suppose we define performance in terms of speed. This still leaves two possible definitions. You could define the fastest plane as the one with the highest cruising speed, taking a single passenger from one point to another in the least time. If you were interested in transporting 450 passengers from one point to another, how-

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers x m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

FIGURE 4.1 The capacity, range, and speed for a number of commercial airplanes. The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times).

response time Also called **execution time**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

ever, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a data center that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred to as **execution time**. Data center managers are often interested in increasing *throughput*—the total amount of work done in a given time. Hence, in most cases, we will need different performance metrics as well as different sets of applications to benchmark desktop computers versus servers, and embedded computers require yet other metrics and applications. We will see examples of this in section 4.4 when we look at different SPEC benchmarks: one meant to measure CPU performance (SPEC CPU) and one meant to measure Web server performance (SPECweb99).

EXAMPLE

Throughput and Response Time

Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the World Wide Web.

ANSWER

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases. If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other.

In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. (In Chapter 8, on input/output systems, we will discuss throughput-related measures.) To maximize performance, we want to minimize response time or execution time for some task. Thus we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\begin{aligned} \text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X \end{aligned}$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase “X is n times faster than Y”—or equivalently “X is n times as fast as Y”—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times faster than Y, then the execution time on Y is n times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Relative Performance

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

EXAMPLE

We know that A is n times faster than B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

ANSWER

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times faster than B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

For simplicity, we will normally use the terminology *faster than* when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*, we usually say “improve performance” or “improve execution time” when we mean “increase performance” and “decrease execution time.”

Elaboration: Performance in embedded systems is often characterized by real-time constraints: that is, certain application-specific events must occur within a limited amount of time. There are two common types of real-time constraints: hard real time and soft real time. Hard real time defines a fixed bound on the time to respond to or process some event. For example, the embedded processor that controls an antilock brake system must respond within a hard limit from the time it receives a signal that the wheels are locked. In soft real-time systems, an average response or a response within a limited time to a large fraction of the events suffices. For example, handling video frames in a DVD playback system would be an example of a soft real-time constraint, since dropping a frame is permissible, if it happens very rarely. In embedded real-time applications, once the response-time performance constraint is met, designers often optimize throughput or try to reduce cost.

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. But time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time that the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which

CPU execution time Also called **CPU time**. The actual time the CPU spends computing for a specific task.

recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to do accurately because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems.

For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term *system performance* to refer to elapsed time on an unloaded system, and use *CPU performance* to refer to user CPU time. We will focus on CPU performance in this chapter, although our discussions of how to summarize performance can be applied to either elapsed time or to CPU time measurements.

Although as computer users we care about time, when we examine the details of a computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that runs at a constant rate and determines when events take place in the hardware. These discrete time intervals are called **clock cycles** (or ticks, clock ticks, clock periods, clocks, cycles). Designers refer to the length of a **clock period** both as the time for a complete *clock cycle* (e.g., 0.25 nanoseconds, 0.25 ns, 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next section, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

user CPU time The CPU time spent in a program itself.

system CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

clock cycle Also called tick, clock tick, clock period, clock, cycle. The time for one clock period, usually of the processor clock, which runs at a constant rate.

clock period The length of each clock cycle.

Different applications are sensitive to different aspects of the performance of a computer system. Many applications, especially those running on servers, depend as much on I/O performance, which, in turn, relies on both hardware and software, and total elapsed time measured by a wall clock is the measurement of interest. In some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to look for the performance bottlenecks by measuring program execution and looking for the likely bottlenecks. In the following chapters, we will describe how to search for bottlenecks and improve performance in various parts of the system.

Understanding Program Performance

Check Yourself

1. Suppose we know that an application that uses both a desktop client and a remote server is limited by network performance. For the following changes state whether only the throughput improves, both response time and throughput improve, or neither improves.
 - a. An extra network channel is added between the client and the server, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).
 - b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.
 - c. More memory is added to the computer.
2. Computer C's performance is 4 times better than the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

4.2

CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as seen by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program. As we will see in this chapter and later in Chapters 5, 6, and 7, the designer often faces a trade-off between the number of clock cycles

needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles also increase the clock cycle time.

Improving Performance

Our favorite program runs in 10 seconds on computer A, which has a 4 GHz clock. We are trying to help a computer designer build a computer, B, that will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

Let's first find the number of clock cycles required for the program on A:

$$\begin{aligned}\text{CPU time}_A &= \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A} \\ 10 \text{ seconds} &= \frac{\text{CPU clock cycles}_A}{4 \times 10^9 \frac{\text{cycles}}{\text{second}}}\end{aligned}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 4 \times 10^9 \frac{\text{cycles}}{\text{second}} = 40 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\begin{aligned}\text{CPU time}_B &= \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B} \\ 6 \text{ seconds} &= \frac{1.2 \times 40 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}\end{aligned}$$

$$\text{Clock rate}_B = \frac{1.2 \times 40 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{8 \times 10^9 \text{ cycles}}{\text{second}} = 8 \text{ GHz}$$

computer B must therefore have twice the clock rate of A to run the program in 6 seconds.

EXAMPLE

ANSWER

Hardware Software Interface

The equations in our previous examples do not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \frac{\text{Average clock cycles}}{\text{per instruction}}$$

clock cycles per instruction (CPI) Average number of clock cycles per instruction for a program or program fragment.

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the instruction count required for a program will, of course, be the same.

EXAMPLE

ANSWER

Using the Performance Equation

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program, and by how much?

We know that each computer executes the same number of instructions for the program; let's call this number I . First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time}_A \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters

How can we determine the value of these factors in the performance equation? We can measure the CPU execution time by running the program, and the clock cycle time is usually published as part of the documentation for a computer. The instruction count and CPI can be more difficult to obtain. Of course, if we know the clock rate and CPU execution time, we need only one of the instruction count or the CPI to determine the other.

We can measure the instruction count by using software tools that profile the execution or by using a simulator of the architecture. Alternatively, we can use hardware counters, which are included on many processors, to record a variety of measurements, including the number of instructions executed, the average CPI, and often, the sources of performance loss. Since the instruction count depends

The **BIG**
Picture

Figure 4.2 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds per program:

$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time that offsets the improvement in instruction count. Similarly, because CPI depends on type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

FIGURE 4.2 The basic components of performance and how each is measured.

on the architecture, but not on the exact implementation, we can measure the instruction count without knowing all the details of the implementation. The CPI, however, depends on a wide variety of design details in the computer, including both the memory system and the processor structure (as we will see in Chapters 5, 6, and 7), as well as on the mix of instruction types executed in an application. Thus, CPI varies by application, as well as among implementations with the same instruction set.

Designers often obtain CPI by a detailed simulation of an implementation or by using hardware counters, when a CPU is operational. Sometimes it is possible to compute the CPU clock cycles by looking at the different types of instructions and using their individual clock cycle counts. In such cases, the following formula is useful:

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

where C_i is the count of the number of instructions of class i executed, CPI_i is the average number of cycles per instruction for that instruction class, and n is the number of instruction classes. Remember that overall CPI for a program will depend on both the number of cycles for each instruction type and the frequency of each instruction type in the program execution.

As we described in Chapter 1, the performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Understanding Program Performance

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher-CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

EXAMPLE

Comparing Code Segments

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level-language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

ANSWER

Sequence 1 executes $2 + 1 + 2 = 5$ instructions. Sequence 2 executes $4 + 1 + 1 = 6$ instructions. So sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it actually executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

The above example shows the danger of using only one factor (instruction count) to assess performance. When comparing two computers, you must look at all three components, which combine to form execution time. If some of the factors are identical, like the clock rate in the above example, performance can be determined by comparing all the nonidentical factors. Since CPI varies by **instruction mix**, both instruction count and CPI must be compared, even if clock rates are identical. Several of the exercises ask you to evaluate a series of computer and compiler enhancements that affect clock rate, CPI, and instruction count. In the next section, we'll examine a common performance measurement that does not incorporate all the terms and can thus be misleading.

Two of the major factors that affect CPI are the performance of the pipeline, which is the technique used by all modern processors to execute instructions, and the performance of the memory system. In Chapter 6, we will see how pipeline performance adds to the CPI through stalls, and in Chapter 7 we will see how the performance of the caches can increase the CPI due to stalls in the memory system.

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler?

- a. $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$
- b. $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$
- c. $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

instruction mix A measure of the dynamic frequency of instructions across one or many programs.

Check Yourself

4.3

Evaluating Performance

workload A set of programs run on a computer that is either the actual collection of applications run by a user or is constructed from real programs to approximate such a mix. A typical workload specifies both the programs as well as the relative frequencies.

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. The set of programs run would form a **workload**. To evaluate two computer systems, a user would simply compare the execution time of the workload on the two computers. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate computer, hoping that the methods will reflect how well the computer will perform with the user's workload. This alternative is usually followed by evaluating the computer using a set of *benchmarks*—programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload.

Today, it is widely understood that the best type of programs to use for benchmarks are real applications. These may be applications that the user employs regularly or simply applications that are typical. For example, in an environment where the users are primarily engineers, you might use a set of benchmarks containing several typical engineering or scientific applications. If the user community were primarily software development engineers, the best benchmarks would probably include such applications as a compiler or document processing system. Using real applications as benchmarks makes it much more difficult to find trivial ways to speed up the execution of the benchmark. Furthermore, when techniques are found to improve performance, such techniques are much more likely to help other programs in addition to the benchmark.

The use of benchmarks whose performance depends on very small code segments encourages optimizations in either the architecture or compiler that target these segments. The compiler optimizations might recognize special code fragments and generate an instruction sequence that is particularly efficient for this code fragment. Likewise, a designer might try to make some sequence of instructions run especially fast because the sequence occurs in a benchmark. In fact, several companies have introduced compilers with special-purpose optimizations targeted at specific benchmarks. Often these optimizations must be explicitly enabled with a specific compiler option, which would not be used when compiling other programs. Whether the compiler would produce good code, or even *correct* code, if a real application program used these switches, is unclear.

Sometimes in the quest to produce highly optimized code for benchmarks, engineers introduce erroneous optimizations. For example, in late 1995, Intel published a new performance rating for the integer SPEC benchmarks running on a Pentium processor and using an internal compiler, not used outside of Intel.

Unfortunately, the code produced for one of the benchmarks was wrong, a fact that was discovered when a competitor read through the binary to understand how Intel had sped up one of the programs in the benchmark suite so dramatically. In January 1996, Intel admitted the error and restated the performance. Small programs or programs that spend almost all their execution time in a very small code fragment are especially vulnerable to such efforts.

So why doesn't everyone run real programs to measure performance? One reason is that small benchmarks are attractive when beginning a design, since they are small enough to compile and simulate easily, sometimes by hand. They are especially tempting when designers are working on a novel computer because compilers may not be available until much later in the design. Although the use of such small benchmarks early in the design process may be justified, there is no valid rationale for using them to evaluate working computer systems.

As mentioned earlier, different classes and applications of computers will require different types of benchmarks. For desktop computers, the most common benchmarks are either measures of CPU performance or benchmarks focusing on a specific task, such as DVD playback or graphics performance for games. In Section 4.4, we will examine the SPEC CPU benchmarks, which focus on CPU performance and measure response time to complete a benchmark. For servers, the decision of which benchmark to use depends heavily on the nature of the intended application. For scientific servers, CPU-oriented benchmarks with scientific applications are typically used, and response time to complete a benchmark is the metric. For other server environments, benchmarks of Web serving, file serving, and databases are commonly used. These server benchmarks usually emphasize throughput, albeit with possible requirements on response time to individual events, such as a database query or Web page request. Section 4.4 examines the SPECweb99 benchmark designed to test Web server performance. In embedded computing, good benchmarks are much more rare. Often customers use their specific embedded application or a segment of it for benchmarking purposes. The one major benchmark suite developed for embedded computers is EEMBC, and we discuss those benchmarks in the In More Depth section on the CD.

Once we have selected a set of suitable benchmarks and obtained performance measurements, we can write a performance report. The guiding principle in reporting performance measurements should be *reproducibility*—we should list everything another experimenter would need to duplicate the results. This list must include the version of the operating system, compilers, and the input, as well as the computer configuration. As an example, the system description section of a SPEC CPU2000 benchmark report is in Figure 4.3.

One important element of reproducibility is the choice of input. Different inputs can generate quite different behavior. For example, an input can trigger certain execution paths that may be typical, or it may exercise rarely used, and

Hardware	
Hardware vendor	Dell
Model number	Precision WorkStation 360 (3.2 GHz Pentium 4 Extreme Edition)
CPU	Intel Pentium 4 (800 MHz system bus)
CPU MHz	3200
FPU	Integrated
CPU(s) enabled	1
CPU(s) orderable	1
Parallel	No
Primary cache	12K(I) micro-ops + 8KB(D) on chip
Secondary cache	512KB(I+D) on chip
L3 cache	2048KB(I+D) on chip
Other cache	N/A
Memory	4 x 512MB ECC DDR400 SDRAM CL3
Disk subsystem	1x 80GB ATA/100 7200 RPM
Other hardware	

Software	
Operating system	Windows XP Professional SP1
Compiler	Intel C++ Compiler 7.1 (20030402Z) Microsoft Visual Studio.NET (7.0.9466) MicroQuill SmartHeap Library 6.01
File system type	NTFS
System state	Default

FIGURE 4.3 System description of a desktop system using the fastest Pentium 4 available in 2003. In addition to this formatted mandatory description, there are 23 lines of notes describing special flag settings used for portability (4), optimization (2), tuning (12), base timing (2), a special library (2), and BIOS setting (1).

hence less important, parts of an application. Some of the most important effects from the input set are in the memory system. Larger input sets tend to stress the memory system to a greater extent, and the use of realistically sized workloads in servers both for commercial and scientific applications is critical if a benchmark is intended to predict what real applications may see.

Comparing and Summarizing Performance

Once we have selected programs to use as benchmarks and agreed on whether we are measuring response time or throughput, you might think that performance comparison would be straightforward. However, we must still decide how to summarize the performance of a group of benchmarks. Although summarizing a set of

	Computer A	Computer B
Program 1 (seconds)	1	10
Program 2 (seconds)	1000	100
Total time (seconds)	1001	110

FIGURE 4.4 Execution times of two programs on two different computers. Taken from Figure 1 of Smith [1988].

measurements results in less information, marketers and even users often prefer to have a single number to compare performance. The key question is, How should a summary be computed? Figure 4.4, which is abstracted from an article about summarizing performance, illustrates some of the difficulties facing such efforts.

Using our definition of *faster*, the following statements hold for the program measurements in Figure 4.4:

- A is 10 times faster than B for program 1.
- B is 10 times faster than A for program 2.

Taken individually, each of these statements is true. Collectively, however, they present a confusing picture—the relative performance of computers A and B is unclear.

Total Execution Time: A Consistent Summary Measure

The simplest approach to summarizing relative performance is to use total execution time of the two programs. Thus

$$\frac{\text{Performance}_B}{\text{Performance}_A} = \frac{\text{Execution time}_A}{\text{Execution time}_B} = \frac{1001}{110} = 9.1$$

That is, B is 9.1 times as fast as A for programs 1 and 2 together.

This summary is directly proportional to execution time, our final measure of performance. If the workload consists of running programs 1 and 2 an equal number of times, this statement would predict the relative execution times for the workload on each computer.

The average of the execution times that is directly proportional to total execution time is the **arithmetic mean** (AM):

$$\text{AM} = \frac{1}{n} \sum_{i=1}^n \text{Time}_i$$

arithmetic mean The average of the execution times that is directly proportional to total execution time.

weighted arithmetic mean

An average of the execution time of a workload with weighting factors designed to reflect the presence of the programs in a workload; computed as the sum of the products of weighting factors and execution times.

Check Yourself

where $Time_i$ is the execution time for the i th program of a total of n in the workload. Since it is the mean of execution times, a smaller mean indicates a smaller average execution time and thus improved performance.

The arithmetic mean is proportional to execution time, assuming that the programs in the workload are each run an equal number of times. Is that the right workload? If not, we can assign a weighting factor w_i to each program to indicate the frequency of the program in that workload. If, for example, 20% of the tasks in the workload were program 1 and 80% of the tasks in the workload were program 2, then the weighting factors would be 0.2 and 0.8. By summing the products of weighting factors and execution times, we can obtain a clear picture of the performance of the workload. This sum is called the **weighted arithmetic mean**. One method of weighting programs is to choose weights so that the execution time of each benchmark is equal on the computer used as the base. The standard arithmetic mean is a special case of the weighted arithmetic mean where all weights are equal. We explore the weighted mean in more detail in Exercises 4.15 and 4.16.

1. Suppose you are choosing between four different desktop computers: one is an Apple MacIntosh and the other three are PC-compatible computers that use a Pentium 4, an AMD processor (using the same compiler as the Pentium 4), and a Pentium 5 (which does not yet exist in 2004 but has the same architecture as the Pentium 4 and uses the same compiler). Which of the following statements are true?
 - a. The fastest computer will be the one with the highest clock rate.
 - b. Since all PCs use the same Intel-compatible instruction set and execute the same number of instructions for a program, the fastest PC will be the one with the highest clock rate.
 - c. Since AMD uses different techniques than Intel to execute instructions, they may have different CPIs. But, you can still tell which of the two Pentium-based PCs is fastest by looking at the clock rate.
 - d. Only by looking at the results of benchmarks for tasks similar to your workload can you get an accurate picture of likely performance.
2. Assume the following measurements of execution time were taken:

Program	Computer A	Computer B
1	2 sec	4 sec
2	5 sec	2 sec

Which of the following statements is true?

- a. A is faster than B for program 1.
- b. A is faster than B for program 2.
- c. A is faster than B for a workload with equal numbers of executions of program 1 and 2.
- d. A is faster than B for a workload with twice as many executions of program 1 as of program 2.

4.4

Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors

SPEC (System Performance Evaluation Corporation) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems. It began in 1989 focusing on benchmarking workstations and servers using CPU-intensive benchmarks. (A more detailed history is contained in Section 4.7.) Today, SPEC offers a dozen different benchmark sets designed to test a wide variety of computing environments using real applications and strictly specified execution rules and reporting requirements. The SPEC benchmark sets include benchmarks for CPU performance, graphics, high-performance computing, object-oriented computing, Java applications, client-server models, mail systems, file systems, and Web servers. In this section, we examine the performance of a variety of Dell computer systems that use Pentium III and Pentium 4 processors using a CPU performance benchmark and a Web-oriented system benchmark.

Performance with SPEC CPU Benchmarks

The latest release of the [SPEC CPU benchmarks](#) is the SPEC CPU2000 suite, which consists of 12 integer and 14 floating-point programs, as shown in Figure 4.5. The SPEC CPU benchmarks are intended to measure CPU performance, although wall clock time is the reported measurement. Separate summaries are reported for the integer and floating-point benchmark sets. The execution time measurements are first normalized by dividing the execution time on a Sun Ultra 5_10 with a 300 MHz processor by the execution time on the measured computer; this normalization yields a measure, called the *SPEC ratio*, which has the advantage that bigger numeric results indicate faster performance (i.e., SPEC ratio is the inverse of execution time). A CINT2000 or CFP2000 summary measurement is obtained by taking the geometric mean of the SPEC ratios. (See the In More Depth section on the CD for a discussion of trade-offs in using geometric mean.)

system performance evaluation cooperative (SPEC) benchmark A set of standard CPU-intensive, integer and floating point benchmarks based on real programs.

Integer benchmarks		FP benchmarks	
Name	Description	Name	Type
gzip	Compression	wupwise	Quantum chromodynamics
vpr	FPGA circuit placement and routing	swim	Shallow water model
gcc	The Gnu C compiler	mgrid	Multigrid solver in 3-D potential field
mcf	Combinatorial optimization	applu	Parabolic/elliptic partial differential equation
crafty	Chess program	mesa	Three-dimensional graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition using neural networks
perlbnk	perl application	equake	Seismic wave propagation simulation
gap	Group theory, interpreter	facerec	Image recognition of faces
vortex	Object-oriented database	ammp	Computational chemistry
bzip2	Compression	lucas	Primality testing
twolf	Place and rote simulator	fma3d	Crash simulation using finite-element method
		sixtrack	High-energy nuclear physics accelerator design
		apsi	Meteorology: pollutant distribution

FIGURE 4.5 The SPEC CPU2000 benchmarks. The 12 integer benchmarks in the left half of the table are written in C and C++, while the floating-point benchmarks in the right half are written in Fortran (77 or 90) and C. For more information on SPEC and on the SPEC benchmarks, see www.spec.org. The SPEC CPU benchmarks use wall clock time as the metric, but because there is little I/O, they measure CPU performance.

For a given instruction set architecture, increases in CPU performance can come from three sources:

1. Increases in clock rate
2. Improvements in processor organization that lower the CPI
3. Compiler enhancements that lower the instruction count or generate instructions with a lower average CPI (e.g., by using simpler instructions)

To illustrate such performance improvements, Figure 4.6 shows the SPEC CINT2000 and CFP2000 measurements for a series of Intel Pentium III and Pentium 4 processors measured using the Dell Precision desktop computers. Since SPEC requires that the benchmarks be run on real hardware and the memory system has a significant effect on performance, other systems with these processors may produce different performance levels. In addition to the clock rate differences, the Pentium III and Pentium 4 use different pipeline structures, which we describe in more detail in Chapter 6.

There are several important observations from these two performance graphs. First, observe that the performance of each of these processors scales almost linearly with clock rate increases. Often this is not the case, since losses in the mem-

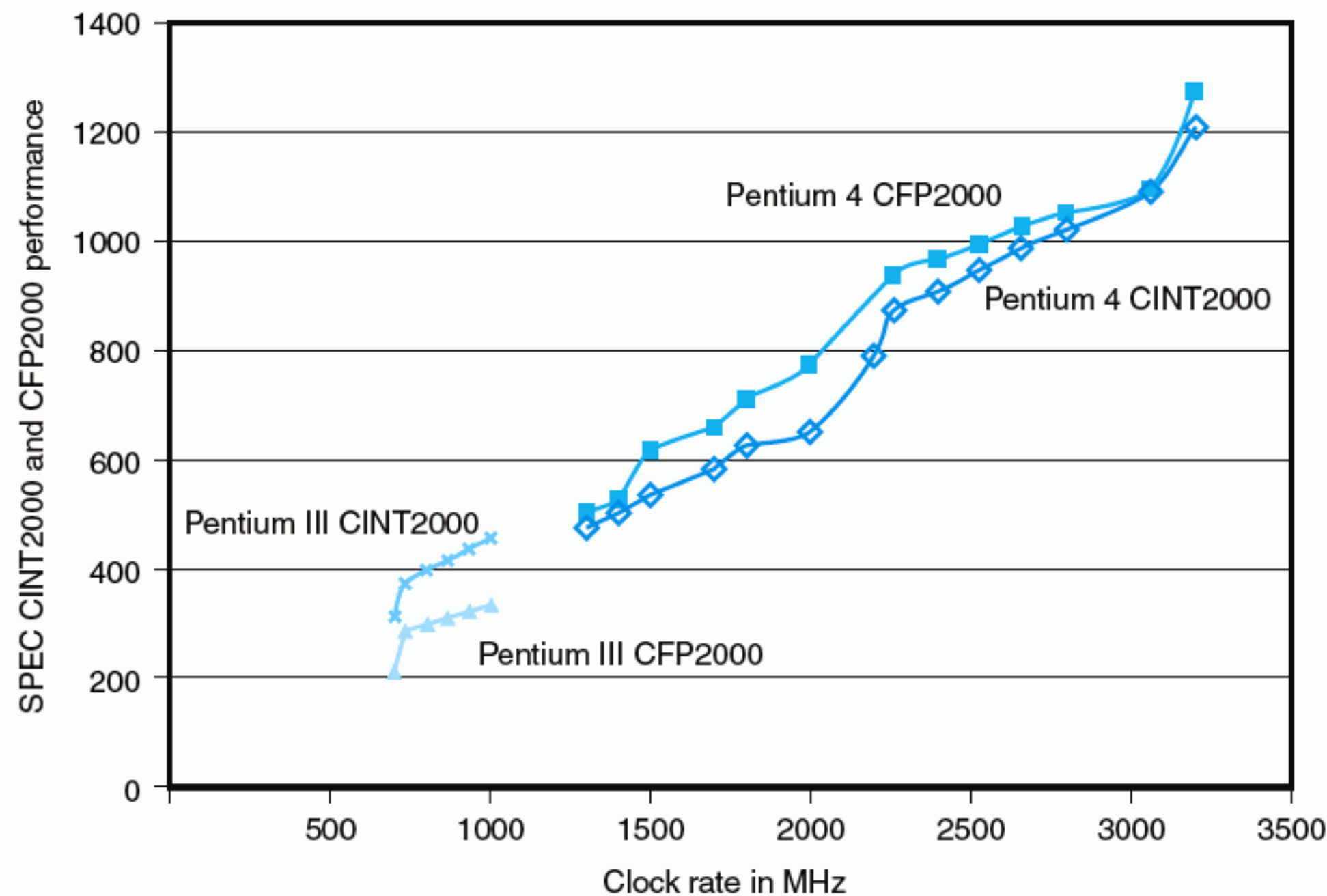


FIGURE 4.6 The SPEC CINT2000 and CFP2000 ratings for the Intel Pentium III and Pentium 4 processors at different clock speeds. SPEC requires two sets of measurements: one that allows aggressive optimization with benchmark-specific switches, and one that allows only the standard optimization switches (called the “base” measurements). This chart contains base measurements; for the integer benchmarks the differences are small. For more details on SPEC, see www.spec.org. The Pentium 4 Extreme (a version of the Pentium 4 introduced in late 2003) is not included in these results, since it uses a different cache architecture than the rest of the Pentium 4 processors.

ory system, which we discuss in Chapter 7, often worsen with higher clock rates. The strong performance of these processors is due both to the aggressive cache systems used in these processors, and the inability of many of the SPEC benchmarks to stress such a memory system.

Comparing the Pentium III and Pentium 4 performances yields even more interesting insights. In particular, note the relative positions of the CINT2000 and CFP2000 curves for the Pentium III versus the Pentium 4. One can quickly draw the inference that the Pentium 4 is either relatively better than the Pentium III on floating-point benchmarks or relatively worse on integer benchmarks. But, which is the case?

The Pentium 4 uses a more advanced integrated circuit technology as well as a more aggressive pipeline structure, both of which allow for a significant clock rate

increase. One comparison that is particularly interesting is the SPEC CINT2000 and CFP2000 measurement divided by the clock rate in MHz. The following tables summarizes the average value of this ratio across different clock rates for each processor:

Ratio	Pentium III	Pentium 4
CINT2000 / Clock rate in MHz	0.47	0.36
CFP2000 / Clock rate in MHz	0.34	0.39

Metrics such as benchmark performance divided by clock rate are sometimes thought of as measurements of implementation efficiency, although, as we have seen, one cannot separate the interaction of clock rate and other improvements.

These measurements are particularly interesting because of the differences between the integer and floating-point benchmarks. The CINT2000 performance ratios are typical: when a faster version of a processor is introduced it may sacrifice one aspect of a design (such as CPI) to enhance another (such as clock rate). Assuming one compiler for both processors, and hence identical code, the CINT2000 ratios tell us that the CPI of the Pentium 4 is 1.3 (0.47/0.36) times that of the Pentium 3.

How then can these numbers be reversed for the floating-point benchmarks? The answer is that the Pentium 4 provides a set of new instructions (called the Streaming SIMD Extensions 2; see Chapter 3) that provide a significant boost for floating point. Thus, both the instruction count and the CPI for the Pentium 4 will differ from that of the Pentium 3, producing improved performance.

SPECweb99: A Throughput Benchmark for Web Servers

In 1996, SPEC introduced its first benchmark designed to measure Web server performance; the benchmark was superseded by a new version in 1999. SPECweb99 differs from SPEC CPU in a number of ways:

- SPECweb99 focuses on throughput, measuring the maximum number of connections that a system running as a Web server can support. The system must provide response to a client request within a bounded time and with a bounded number of errors.
- Because SPECweb99 measures throughput, multiprocessors (systems with more than one CPU) are often used in benchmarks.
- SPECweb99 provides only a program to generate Web server requests; the Web server software becomes part of the system being measured.
- SPECweb99 performance depends on a wide measure of system characteristics, including the disk system and the network.

System	Processor	Number of disk drives	Number of CPUs	Number of networks	Clock rate (GHz)	Result
1550/1000	Pentium III	2	2	2	1	2765
1650	Pentium III	3	2	1	1.4	1810
2500	Pentium III	8	2	4	1.13	3435
2550	Pentium III	1	2	1	1.26	1454
2650	Pentium 4 Xeon	5	2	4	3.06	5698
4600	Pentium 4 Xeon	10	2	4	2.2	4615
6400/700	Pentium III Xeon	5	4	4	0.7	4200
6600	Pentium 4 Xeon MP	8	4	8	2	6700
8450/700	Pentium III Xeon	7	8	8	0.7	8001

FIGURE 4.7 SPECweb9999 performance for a variety of Dell PowerEdge systems using the Xeon versions of the Pentium III and Pentium 4 microprocessors.

To show how these characteristics produce a remarkably varied picture of Web server performance, we selected the SPECweb99 results for a series of Dell PowerEdge servers that use the Xeon versions of the Pentium III and Pentium 4 processors. The Xeon processors are built using the basic structure of the Pentium III or Pentium 4, but support multiprocessing. In addition the Xeon MP supports a third level of off-chip cache and can support more than two processors. The results for a variety of these Dell systems are shown in Figure 4.7.

Looking at the data in Figure 4.7, we can see that clock rate of the processors is clearly not the most important factor in determining Web server performance. In fact, the 8400 has twice as many slow processors as the 6600 and yet offers better performance. We expect these systems to be configured to achieve the best performance. That is, for a given set of CPUs, disks and additional networks are added until the processor becomes the bottleneck.

Performance, Power, and Energy Efficiency

As mentioned in Chapter 1, power is increasingly becoming the key limitation in processor performance. In the embedded market, where many processors go into environments that rely solely on passive cooling or on battery power, power consumption is often a constraint that is as important as performance and cost.

No doubt, many readers will have encountered power limitations when using their laptops. Indeed, between the challenges of removing excess heat and the limitations of battery life, power consumption has become a critical factor in the

design of processors for laptops. Battery capacity has improved only slightly over time, with the major improvements coming from new materials. Hence, the ability of the processor to operate efficiently and conserve power is crucial. To save power, techniques ranging from putting parts of the computer to sleep, to reducing clock rate and voltage, have all been used. In fact, power consumption is so important that Intel has designed a line of processors, the Pentium M series, specifically for mobile, battery-powered applications.

As we discussed in Chapter 1, for CMOS technology, we can reduce power by reducing frequency. Hence, recent processors intended for laptop use all have the ability to adapt frequency to reduce power consumption, simultaneously, of course, reducing performance. Thus, adequately evaluating the energy efficiency of a processor requires examining its performance at maximum power, at an intermediate level that conserves battery life, and at a level that maximizes battery life. In the Intel Mobile Pentium and Pentium M lines, there are two available clock rates: maximum and a reduced clock rate. The best performance is obtained by running at maximum speed, the best battery life by running always at the reduced rate, and the intermediate, performance-power optimized level by switching dynamically between these two clock rates.

Figure 4.8 shows the performance of three Intel Pentium processors designed for use in mobile applications running SPEC CINT2000 and SPEC CFP2000 as benchmarks. As we can see, the newest processor, the Pentium M, has the best performance when run a full clock speed, as well as with the adaptive clock rate mode. The Pentium M's 600 MHz clock makes it slower when run in minimum power mode than the Pentium 4-M, but still faster than the older Pentium III-M design.

For power-limited applications, the most important metric is probably energy efficiency, which is computed by taking performance and dividing by average power consumption when running the benchmark. Figure 4.9 shows the relative energy efficiency for these processors running the SPEC2000 benchmarks. This data clearly shows the energy efficiency advantage of the newer Pentium M design. In all three modes, it has a significant advantage in energy efficiency over the mobile versions of the Pentium III and Pentium 4. Notice that the Pentium 4-M has only a slight efficiency advantage over the Pentium III-M. This data clearly shows the advantage of a processor like the Pentium M, which is designed for reduced power usage from the start, as opposed to a design like the Pentium III-M or Pentium 4-M, which are modified versions of the standard processors. Of course, adequately measuring energy efficiency also requires the use of additional benchmarks designed to reflect how users employ battery-powered computers. Both PC review magazines and Intel's technical journal regularly undertake such studies.

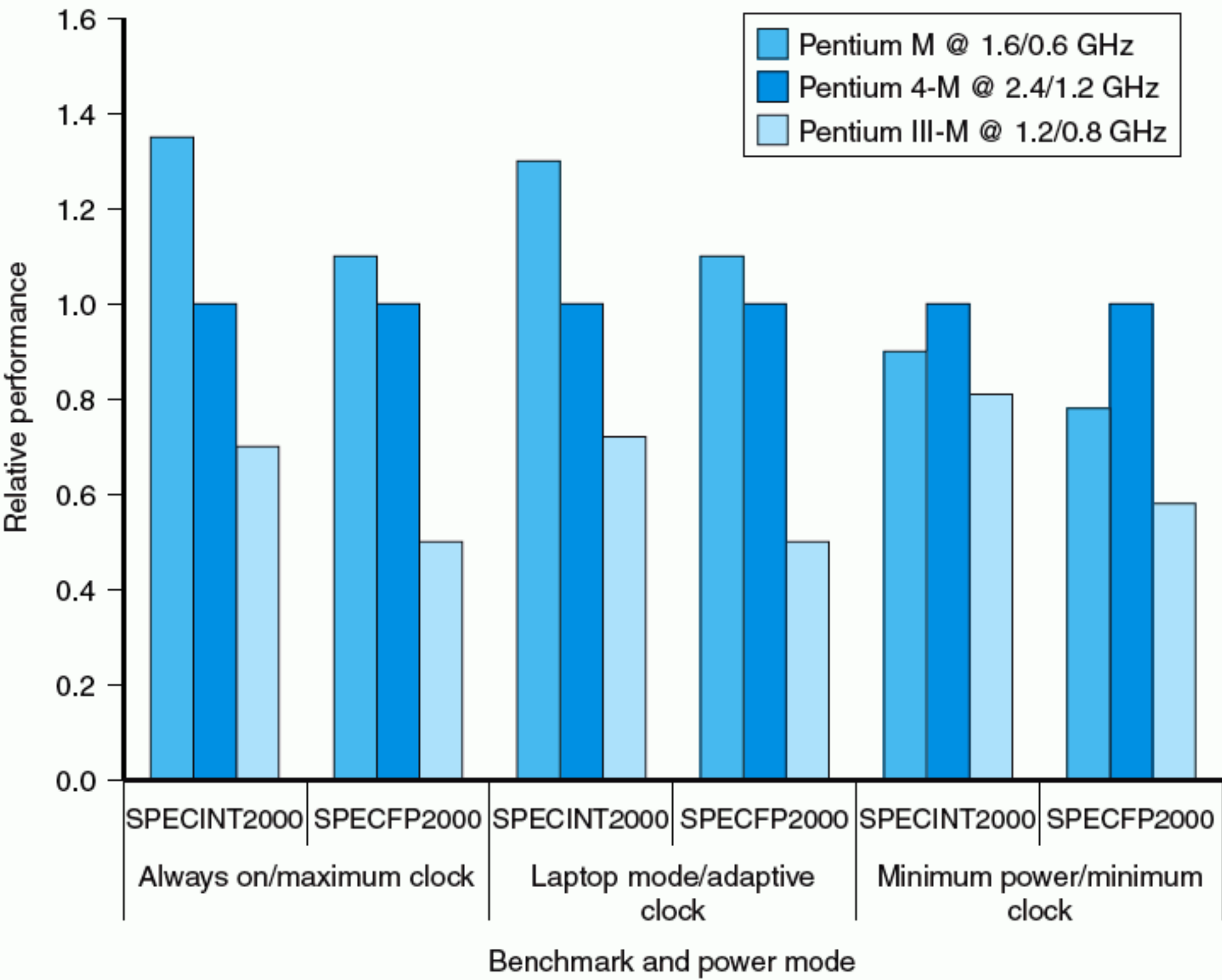


FIGURE 4.8 Relative performance of three Intel processors on SPECINT2000 and SPECFP2000 in three different modes. Each processor operates at two different clock rates, listed in the legend.

Which of the following one-processor Pentium III configurations is likely to produce the best performance on SPECweb99 based on the data in Figure 4.7?

- a. 1.26 GHz processor, 1 disk, 1 network connections
- b. 1 GHz processor, 6 disks, 3 network connections
- c. 1.1 GHz processor, 2 disks, 2 network connections

Check Yourself

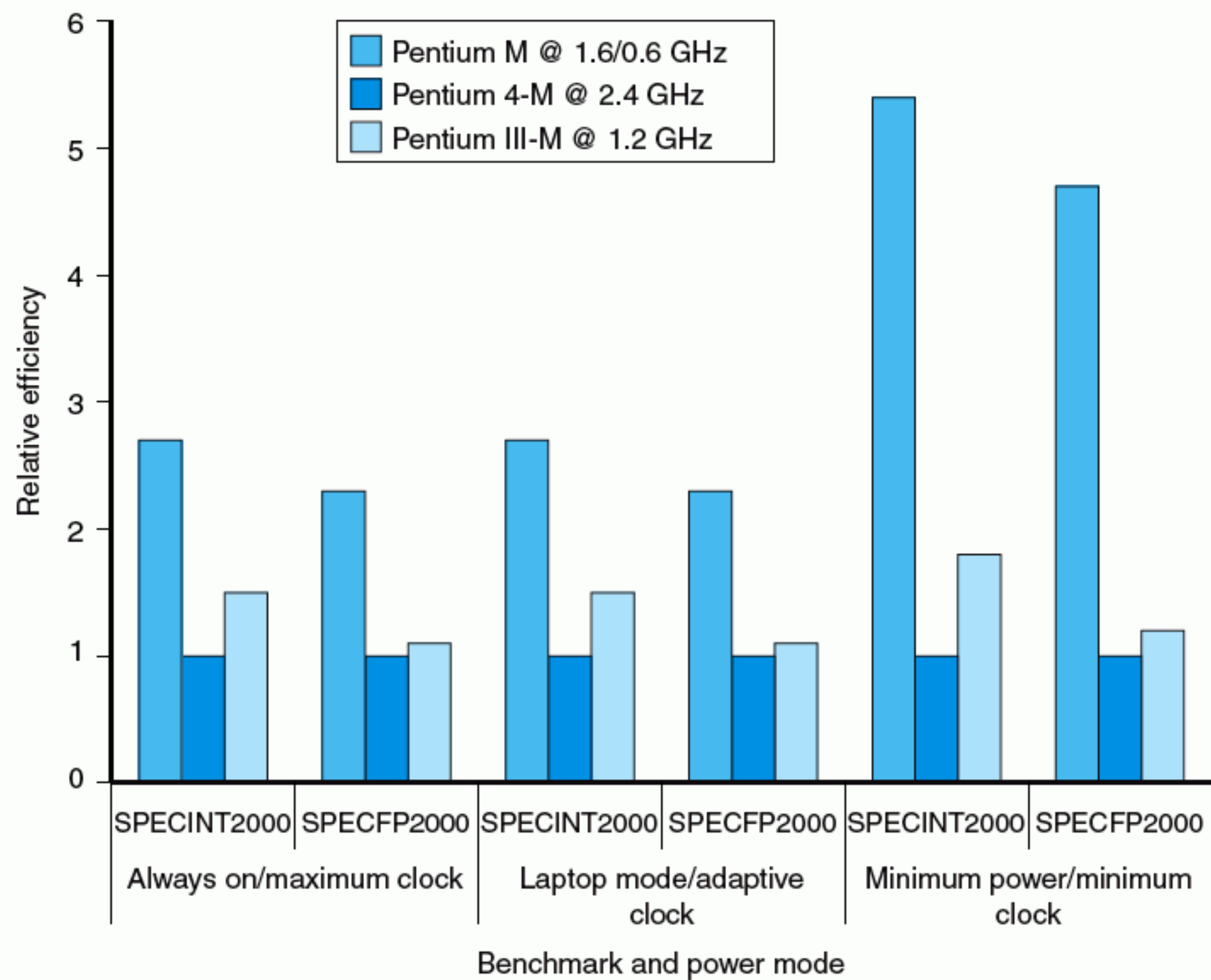


FIGURE 4.9 The relative energy efficiency of three mobile Pentium processors running SPEC2000 in three modes. Energy efficiency is measured as the inverse of joules consumed per benchmark, which is computed by dividing the inverse of the execution time for a benchmark by the watts dissipated.

4.5

Fallacies and Pitfalls

Cost/performance fallacies and pitfalls have ensnared many a computer architect, including us. Accordingly, this section suffers no shortage of relevant examples. We start with a pitfall that has trapped many designers and reveals an important relationship in computer design.

Pitfall: Expecting the improvement of one aspect of a computer to increase performance by an amount proportional to the size of the improvement.

This pitfall has visited designers of both hardware and software. A simple design problem illustrates it well. Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the speed of multiplication if I want my program to run five times faster?

The execution time of the program after I make the improvement is given by the following simple equation known as **Amdahl's law**:

$$\begin{aligned} & \text{Execution time after improvement} \\ &= \left(\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected} \right) \end{aligned}$$


For this problem:

$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$\begin{aligned} 20 \text{ seconds} &= \frac{80 \text{ seconds}}{n} + 20 \text{ seconds} \\ 0 &= \frac{80 \text{ seconds}}{n} \end{aligned}$$

That is, there is *no amount* by which we can enhance multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

This concept also yields what we call the law of diminishing returns in everyday life. We can use Amdahl's law to estimate performance improvements when we know the time consumed for some function and its potential speed up. Amdahl's law, together with the CPU performance equation, are handy tools for evaluating potential enhancements. Amdahl's law is explored in more detail in the exercises and in the  **In More Depth**: Amdahl's Law on the CD.

A common theme in hardware design is a corollary of Amdahl's law: *Make the common case fast*. This simple guideline reminds us that in many cases the frequency with which one event occurs may be much higher than another. Amdahl's law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance.

Pitfall: Using a subset of the performance equation as a performance metric.


We have already shown the fallacy of predicting performance based on simply one of clock rate, instruction count, or CPI. Another common mistake is to use two of the three factors to compare performance. Although using two of the three factors may be valid in a limited context, it is also easily misused. Indeed, nearly all proposed alternatives to the use of time as the performance metric have led eventually to misleading claims, distorted results, or incorrect interpretations.

Amdahl's law A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

million instructions per second (MIPS) A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6 .

One alternative to time as the metric is **MIPS (million instructions per second)**. For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

This MIPS measurement is also called *native MIPS* to distinguish it from some alternative definitions of MIPS that we discuss in  in Section 4.8 on the CD.

Since MIPS is an instruction execution rate, MIPS specifies performance inversely to execution time; faster computers have a higher MIPS rating. The good news about MIPS is that it is easy to understand, and faster computers mean bigger MIPS, which matches intuition.

There are three problems with using MIPS as a measure for comparing computers. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ. In our earlier example examining the SPEC CFP2000 performance, using MIPS to compare the performance of the Pentium III and Pentium 4 would yield misleading results. Second, MIPS varies between programs on the same computer; thus a computer cannot have a single MIPS rating for all programs. Finally and most importantly, MIPS can vary inversely with performance! There are many examples of this anomalous behavior; one is given below.

EXAMPLE

MIPS as a Performance Measure

Consider the computer with three instruction classes and CPI measurements from the last example on page 252. Now suppose we measure the code for the same program from two different compilers and obtain the following data:

Code from	Instruction counts (in billions) for each instruction class		
	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

Assume that the computer’s clock rate is 4 GHz. Which code sequence will execute faster according to MIPS? According to execution time?

First we find the execution time for the two different compilers using the following equation:

ANSWER

$$\text{Execution time} = \frac{\text{CPU clock cycles}}{\text{Clock rate}}$$

We can use an earlier formula for CPU clock cycles:

$$\text{CPU clock cycles} = \sum_{i=1}^n (\text{CPI}_i \times C_i)$$

$$\text{CPU clock cycles}_1 = (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9$$

$$\text{CPU clock cycles}_2 = (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9$$

Now, we find the execution time for the two compilers:

$$\text{Execution time}_1 = \frac{10 \times 10^9}{4 \times 10^9} = 2.5 \text{ seconds}$$

$$\text{Execution time}_2 = \frac{15 \times 10^9}{4 \times 10^9} = 3.75 \text{ seconds}$$

So, we conclude that compiler 1 generates the faster program, according to execution time. Now, let's compute the MIPS rate for each version of the program, using

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$\text{MIPS}_1 = \frac{(5 + 1 + 1) \times 10^9}{2.5 \times 10^6} = 2800$$

$$\text{MIPS}_2 = \frac{(10 + 1 + 1) \times 10^9}{3.75(30) \times 10^6} = 3200$$

So, the code from compiler 2 has a higher MIPS rating, but the code from compiler 1 runs faster!

As examples such as this show, MIPS can fail to give a true picture of performance—even when comparing two versions of the same program on the same computer. In Section 2.7, we discuss other uses of the term *MIPS*, and how such usages can also be misleading.

Check Yourself

Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

- a. Which computer has the higher MIPS rating?
- b. Which computer is faster?

4.6

Concluding Remarks

Although we have focused on performance and how to evaluate it in this chapter, designing only for performance without considering cost, functionality, and other requirements is unrealistic. All computer designers must balance performance and cost. Of course, there exists a domain of *high-performance design*, in which performance is the primary goal and cost is secondary. Much of the supercomputer and high-end server industry designs in this fashion. At the other extreme is *low-cost design*, typified by the embedded market, where cost and power take precedence over performance. In the middle are most desktop designs, as well as low-end servers; these computers require *cost/performance design*, in which the designer balances cost against performance. Examples from the desktop computer industry typify the kinds of trade-offs that designers in this region must live with.

We have seen in this chapter that there is a reliable method of determining and reporting performance, using the execution time of real programs as the metric. This execution time is related to other important measurements we can make by the following equation:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

We will use this equation and its constituent factors many times. Remember, though, that individually the factors do not determine performance: Only the product, which equals execution time, is a reliable measure of performance.

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric valid in a limited context is extended and used beyond that context or without the additional clarification needed to make it valid.

Similarly, any measure that summarizes performance should reflect execution time. Weighted arithmetic means summarize performance while tracking execution time. Through the use of weights, a weighted arithmetic mean can adjust for different running times, balancing the contribution of each benchmark to the summary.

**The BIG
Picture**

Of course, simply knowing this equation is not enough to guide the design or evaluation of a computer. We must understand how the different aspects of a design affect each of these key parameters. This insight involves a wide variety of issues, from the effectiveness of the compiler, to the effects of instruction set design on instruction count, to the impact of pipelining and memory systems on CPI, to the interaction between the technology and organization that determine the clock rate. The art of computer design and evaluation lies not in plugging numbers into a performance equation, but in accurately determining how alternatives will affect performance and cost.

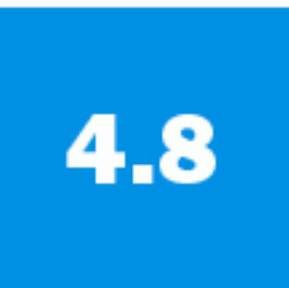
Most computer users care about both cost and performance. While understanding the relationship among aspects of a design and its performance is challenging, determining the cost of various design features is often a more difficult problem. The cost of a computer is affected not only by the cost of the components, but by the costs of labor to assemble the computer, of research and development overhead, of sales and marketing, and of the profit margin. Finally, because of the rapid change in implementation technologies, the most cost-effective choice today is often suboptimal in six months or a year.

Computer designs will always be measured by cost and performance, as well as other important factors such as power, reliability, cost of ownership, and scalability. Although this chapter has focused on performance, the best designs will strike the appropriate balance for a given market among all these factors.



Historical Perspective and Further Reading

This section, which reviews the history of performance measurement and benchmarking, appears as Section 4.7 on the  CD.



Exercises

4.1 [5] <\$4.1> We wish to compare the performance of two different computers: M1 and M2. The following measurements have been made on these computers:

Program	Time on M1	Time on M2
1	2.0 seconds	1.5 seconds
2	5.0 seconds	10.0 seconds

Which computer is faster for each program, and how many times as fast is it?

4.2 [5] <\$4.1> Consider the two computers and programs in Exercise 4.1. The following additional measurements were made:

Program	Instructions executed on M1	Instructions executed on M2
1	5×10^9	6×10^9

Find the instruction execution rate (instructions per second) for each computer when running program 1.

4.3 [5] <\$4.1> Suppose that M1 in Exercise 4.1 costs \$500 and M2 costs \$800. If you needed to run program 1 a large number of times, which computer would you buy in large quantities? Why?

4.4 [10] <\$4.1>  **For More Practice:** Cost-Effective Computing

4.5 [5] <\$4.1>  **For More Practice:** Cost-Effective Computing

4.6 [5] <\$4.1> Another user has the following requirements for the computers discussed in Exercise 4.1: P1 must be executed 1600 times each hour. Any remaining time is used to run P2. If the computer has enough performance to execute program 1 the required number of times per hour, then performance is measured

by the throughput for program 2. Which computer is faster for this workload? Which computer is more cost-effective?

4.7 [10] <§4.2> Suppose you wish to run a program P with 7.5×10^9 instructions on a 5 GHz machine with a CPI of 0.8.

- What is the expected CPU time?
- When you run P, it takes 3 seconds of wall clock time to complete. What is the percentage of the CPU time P received?

4.8 [10] <§4.2> Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set.

P1 has a clock rate of 4 GHz. P2 has a clock rate of 6 GHz. The average number of cycles for each instruction class for P1 and P2 is as follows:

Class	CPI on P1	CPI on P2
A	1	2
B	2	2
C	3	2
D	4	4
E	3	4

Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

4.9 [5] <§§4.1–4.2> If the number of instructions executed in a certain program is divided equally among the classes of instructions in Exercise 4.8 except for class A, which occurs twice as often as each of the others, how much faster is P2 than P1?

4.10 [12] <§4.2> Consider two different implementations, I1 and I2, of the same instruction set. There are three classes of instructions (A, B, and C) in the instruction set. I1 has a clock rate of 6 GHz, and I2 has a clock rate of 3 GHz. The average number of cycles for each instruction class on I1 and I2 is given in the following table:

Class	CPI on M1	CPI on M2	C1 Usage	C2 Usage	C3 Usage
A	2	1	40%	40%	50%
B	3	2	40%	20%	25%
C	5	2	20%	40%	25%

The table also contains a summary of average proportion of instruction classes generated by three different compilers. C1 is a compiler produced by the makers of I1,

C2 is produced by the makers of I2, and the other compiler is a third-party product. Assume that each compiler uses the same number of instructions for a given program but that the instruction mix is as described in the table. Using C1 on both I1 and I2, how much faster can the makers of I1 claim I1 is compared to I2? Using C2, how much faster can the makers of I2 claim that I2 is compared to I1? If you purchase I1, which compiler would you use? If you purchased I2, which compiler would you use? Which computer and compiler would you purchase if all other criteria are identical, including cost?

4.11 [5] <§4.2> Consider program P, which runs on a 1 GHz machine M in 10 seconds. An optimization is made to P, replacing all instances of multiplying a value by 4 (mult X, X,4) with two instructions that set x to $x + x$ twice (add X,X; add X,X). Call this new optimized program P'. The CPI of a multiply instruction is 4, and the CPI of an add is 1. After recompiling, the program now runs in 9 seconds on machine M. How many multiplies were replaced by the new compiler?

4.12 [5] <§4.2> Your company could speed up a Java program on their new computer by adding hardware support for garbage collection. Garbage collection currently comprises 20% of the cycles of the program. You have two possible changes to the machine. The first one would be to automatically handle garbage collection in hardware. This causes an increase in cycle time by a factor of 1.2. The second would be to provide for new hardware instructions to be added to the ISA that could be used during garbage collection. This would halve the number of instruction needed for garbage collections but increase the cycle time by 1.1. Which of these two options, if either, should you choose?

4.13 [5] <§4.2> For the following set of variables, identify all of the subsets that can be used to calculate execution time. Each subset should be minimal; that is, it should not contain any variable that is not needed.

{CPI, clock rate, cycle time, MIPS, number of instructions in program, number of cycles in program}

4.14 [5] <§4.2> The table below shows the number of floating-point operations executed in three different programs and the runtime for those programs on three different computers:

Program	Floating-point operations	Execution time in seconds		
		Computer A	Computer B	Computer C
Program 1	5×10^9	2	5	10
Program 2	20×10^9	20	20	20
Program 3	40×10^9	200	50	15

Which computer is fastest according to total execution time? How many times as fast is it compared to the other two computers?

4.15 [15] <§§4.2, 4.3> One user has told you that the three programs in Exercise 4.14 constitute the bulk of his workload, but he does not run them equally. The user wants to determine how the three computers compare when the workload consists of different mixes of these three programs. (You know you can use the arithmetic mean to find the relative performance.)

Suppose the total number of floating-point operations (FLOPs) executed in the workload is equally divided among the three programs. That is, program 1 runs 8 times for every time program 3 runs, and program 2 runs twice for every time program 3 runs. Find which computer is fastest for this workload and by what factor. How does this compare with the total execution time with equal numbers of program executions?

4.16 [15] <§§4.2, 4.3> An alternative weighting to that of Exercise 4.15 is to assume that equal amounts of time will be spent running each program on one of the computers. Which computer is fastest using the data given for Exercise 4.14 and assuming a weighting that generates equal execution time for each of the benchmark programs on computer A? Which computer is fastest if we assume a weighting that generates equal execution time on computer B? Computer C? Explain the results.

4.17 [5] <§§4.2–4.3> If the clock rates of computers M1 and M2 in Exercise 4.1 are 4 GHz and 6 GHz, respectively, find the clock cycles per instruction (CPI) for program 1 on both computers using the data in Exercises 4.1 and 4.2.

4.18 [5] <§§4.2–4.3> Assuming the CPI for program 2 on each computer in Exercise 4.1 is the same as the CPI for program 1 found in Exercise 4.17, find the instruction count for program 2 running on each computer using the execution times from Exercise 4.1.

4.19 [5] <§4.3>  **In More Depth:** Amdahl's Law

4.20 [10] <§4.3>  **In More Depth:** Amdahl's Law

4.21 [10] <§4.3>  **In More Depth:** Amdahl's Law

4.22 [5] <§4.3>  **In More Depth:** Amdahl's Law

4.23 [5] <§4.3>  **In More Depth:** Amdahl's Law

















4.24 [20] <§4.3>  **In More Depth:** Amdahl's Law

4.25 [5] <§4.3>  **In More Depth:** Choosing the Right Mean

4.26 [15] <§4.3>  **In More Depth:** Choosing the Right Mean

4.27 [3 hours] <§4.3>  **In More Depth:** Synthetic Benchmarks: Attempts to Replicate "Typical" Behavior

4.28 [3 hours] <§4.3>  **In More Depth:** Synthetic Benchmarks: Attempts to Replicate "Typical" Behavior

- 4.29** [4 hours] <§4.3>  **In More Depth:** MIPS, MOPS, and Other FLOPS
- 4.30** [5] <§4.3>  **In More Depth:** MFLOPS as a Performance Metric
- 4.31** [15] <§4.3>  **In More Depth:** MFLOPS as a Performance metric
- 4.32** [4 hours] <§4.3>  **In More Depth:** MFLOPS as a Performance Metric
- 4.33** [5] <§4.3>  **In More Depth:** Embedded Benchmarks
- 4.34** [20] <§§4.2, 4.3>  **In More Depth:** Using Hardware-Independent Metrics to Try to Predict Performance
- 4.35** [10] <§§4.2, 4.3>  **For More Practice:** Analyzing a Processor with Floating-Point Implemented in Hardware or Software
- 4.36** [10] <§§4.2, 4.3>  **For More Practice:** Analyzing a Processor with Floating point implemented in Hardware or Software
- 4.37** [5] <§§4.2, 4.3>  **For More Practice:** Analyzing a Processor with Floating Point Implemented in Hardware or Software
- 4.38** [10] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.39** [5] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.40** [10] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.41** [5] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.42** [5] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.43** [10] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.44** [10] <§§4.2, 4.3>  **For More Practice:** Analyzing Enhancements to a Processor
- 4.45** [5] <§4.3> Assume that multiply instructions take 12 cycles and account for 15% of the instructions in a typical program, and the other 85% of the instructions require an average of 4 cycles for each instruction. What percentage of time does the CPU spend doing multiplication?
- 4.46** [5] <§4.3> Your hardware engineering team has indicated that it would be possible to reduce the number of cycles required for multiplication to 8 in Exercise

4.45, but this will require a 20% increase in the cycle time. Nothing else will be affected by the change. Should they proceed with the modification?

4.47 [10] <§4.4> Look at the current list of SPEC programs in Figure 4.5 on page 260. Does it include applications that match the ways you typically use your computer? What classes of programs are irrelevant or missing? Why do you think they were or were not included in SPEC? What would have to be done to include/exclude such programs in the next SPEC release?

4.48 [5] <§4.4> If benchmark suites are designed to provide a real-world metric for a specific computing task, explain why benchmarks suites need to be updated.

4.49 [5] <§§4.2, 4.3, 4.4> **In More Depth:** The Difficulty with Kernel Benchmarks

4.50 [15] <§§4.2, 4.3, 4.4>  **In More Depth:** The Difficulty with Kernel Benchmarks

4.51 [10] <§§4.1–4.5> Consider the following hypothetical news release:

“The company will unveil the industry’s first 5 GHz version of the chip, which offers a 25% performance boost over the company’s former speed champ, which runs at 4 GHz. The new chip can be plugged into system boards for the older original chip (which ran at 1 GHz) to provide a 70% performance boost.”

Comment on the definition (or definitions) of performance that you believe the company used. Do you think the news release is misleading?

4.52 [indefinite] <§§4.1–4.5> Collect a set of articles that you believe contain incorrect analyses of performance or use misleading performance metrics to try to persuade readers. For example, an article in the *New York Times* (April 20, 1994, p. D1) described a video game player “that will surpass the computing power of even the most powerful personal computers” and presented the following chart to support the argument that “video game computers may be the supercomputers of tomorrow”:

Computer	Approximate number of instructions per second	Price
1975 IBM mainframe	10,000,000	\$10,000,000
1976 Cray-1	160,000,000	\$20,000,000
1979 Digital VAX	1,000,000	\$200,000
1981 IBM PC	250,000	\$3,000
1984 Sun 2	1,000,000	\$10,000
1994 Pentium-chip PC	66,000,000	\$3,000
1995 Sony PCX video game	500,000,000	\$500
1995 Microunity set-top	1,000,000,000	\$500

The article never discussed how the nature of the instructions should impact the definition of “powerful.” For each article you collect, describe why you think it is misleading or incorrect. Good places to look for material include the business or technology sections of newspapers, magazines (both articles and ads), and the Internet (newsgroups and the Web).

Answers to Check Yourself

§4.1, page 246: 1. a: both, b: latency, c: neither. 2. 7 sec.

§4.2, page 253: 6.

§4.3, page 258: 1. F, F, F, T. 2. T, F, F, T

§4.4, page 265: b.

§4.5, page 270: a. Computer A. b. Computer B.

Computers in the Real World

Moving People Faster and More Safely

Problem: Find ways to help transport people quickly while maintaining safety, comfort, and efficiency.

Solution: For more than 20 years, computers have played an increasingly important role in the control of transportation systems including planes, trains, automobiles, and even ships. Essentially all modern transportation systems rely on computers to enhance their safety, comfort, and efficiency. Computers also play a critical role in improving fuel consumption and reducing air pollution. In these two pages, we examine some of the uses of computers in trains and automobiles. In designing pipelines we must prevent hazards, at all costs, and we try to avoid pipeline delays. Computer-controlled trains try to do the same: hazards are life-threatening and can never be permitted, and delays are to be avoided!

The French TGV (*Train à Grande Vitesse*) is one of the fastest train systems in the world, with a typical top speed of 300 km/hour. Traditionally, trains were controlled by an engineer, using a set of track-side lights and signals that tell the engineer to proceed, slow down, or stop. At 300 km/hour, it is difficult to read such signals, and easy to miss one completely, leading to a possible disaster. In addition, some tracks, such as those containing tight

curves, can be unsafe at the higher speeds; since the TGV was designed to run on existing track, finding a fail-safe method to communicate track conditions was critical.

The TGV designers solved this problem by a clever signaling system, called TVM (*Transmission Voie-Machine*), that runs through the rails and is picked up by antennas in the locomotive. The track is divided into blocks, which are typically about 1.5 km in length. Shorter blocks are used when track conditions change quickly or where a higher degree of safety is critical, such as in the Chunnel, where the block length is about one-tenth as long. Transmitters at the beginning of each block are used to communicate instructions to the cab, where they are carried out by an engineer; a computer also watches the communications, and implements the commands if the engineer fails to.



The Eurostar TGV train in Nice, France.

One challenge is that the stopping distance for the TGV is nominally four blocks (somewhat shorter in an absolute emergency). The time to travel four blocks is 1.2 minutes, and the newest signaling system trains are run at a 3 minute headway, even in fog! So the system must monitor the presence of all trains and ensure the most important property: only one train ever occupies a block of track at the same time! The system constantly communicates the maximum safe speed for the current block, improving performance and safety.

The TVM system has been built with extensive attention to safety, which means heavy use of redundancy to ensure the operation of the system in the face of failure of a component. The failure rate of TVM has been estimated at less than 1 failure in a million years. This attention to safety has paid off: in over two decades of service, there have been no fatal accidents caused by a TVM failure.

The images on the previous page and below show the Eurostar TGV and the cab and operator's seat.

Computers have also played a key role in making cars safer, more efficient, and less polluting. The modern automobile has dozens of microprocessors controlling everything from braking, to ignition, to air bag deployment.

In the area of safety, airbags and antilock brakes have been two of the most important innovations since the seat belt. Antilock brakes preserve the


ability to steer during the severe braking that might occur in emergency conditions. By detecting wheel lockup and alternately applying and releasing the brakes under computer control, an antilock brake system can avoid wheel lockup.

Airbags use a force sensor to detect severe deceleration, which occurs during a collision. The airbags are deployed by a computer that reads the sensor. The new generation of airbags uses a two-stage deployment: when the deceleration indicates a collision of moderate severity, the airbag inflates more slowly, reducing the possibility of passenger injury from rapid deployment in a moderate collision. Reliability of these safety systems has been enhanced by a computer-controlled test that is run every time the vehicle is started.

Modern ignition systems in automobiles aim to enhance mileage while reducing pollution. Happily these two goals are doubly congruent: enhancing mileage reduces pollution through the use of less fuel, and mileage is enhanced by more effective combustion, which also reduces the emission of partially combusted fuel. Computers control the injection of fuel, the amount of air injected, and the spark timing, which must change as the engine runs faster. Careful control of these elements over the full operating range from 1000 to 6000 RPM and different temperature conditions has helped improve mileage and reduce pollution.



The interior of a Eurostar TGV cab.

To learn more see these references on the  library

“An investigation of the Therac-25 accidents,” Nancy G. Leveson and Clark S. Turner. *IEEE Computer*, 26(7): 18–41, July 1993.

5

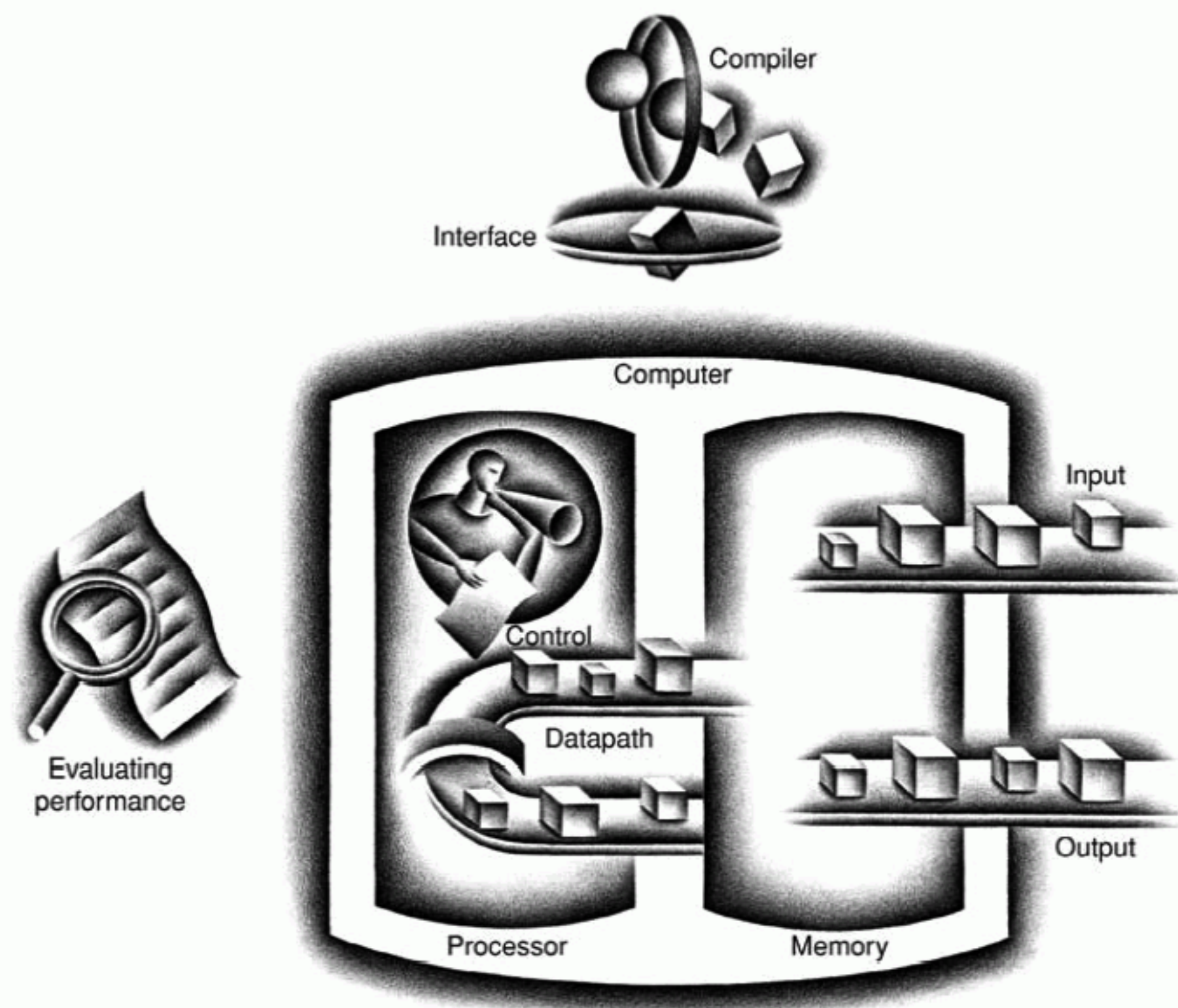
The Processor: Datapath and Control

*In a major matter,
no details are small.*

French Proverb

5.1	Introduction	284
5.2	Logic Design Conventions	289
5.3	Building a Datapath	292
5.4	A Simple Implementation Scheme	300
5.5	A Multicycle Implementation	318
5.6	Exceptions	340
	5.7 Microprogramming: Simplifying Control Design	346
	5.8 An Introduction to Digital Design Using a Hardware Design Language	346
5.9	Real Stuff: The Organization of Recent Pentium Implementations	347
5.10	Fallacies and Pitfalls	350
5.11	Concluding Remarks	352
	5.12 Historical Perspective and Further Reading	353
5.13	Exercises	354

The Five Classic Components of a Computer





5.1 Introduction

In Chapter 4, we saw that the performance of a machine was determined by three key factors: instruction count, clock cycle time, and clock cycles per instruction (CPI). The compiler and the instruction set architecture, which we examined in Chapters 2 and 3, determine the instruction count required for a given program. However, both the clock cycle time and the number of clock cycles per instruction are determined by the implementation of the processor. In this chapter, we construct the datapath and control unit for two different implementations of the MIPS instruction set.

This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview in this section, followed by sections that build up a datapath and construct a simple version of a processor sufficient to implement instructions sets like MIPS, and finally, developing the concepts necessary to implement more complex instructions sets, like the IA-32.

For the reader interested in understanding the high-level interpretation of instructions and its impact on program performance, this initial section provides enough background to understand these concepts as well as the basic concepts of pipelining, which are explained in Section 6.1 of the next chapter.

For those readers desiring an understanding of how hardware implements instructions, Sections 5.3 and 5.4 are all the additional material that is needed. Furthermore, these two sections are sufficient to understand all the material in Chapter 6 on pipelining. Only those readers with an interest in hardware design should go further.

The remaining sections of this chapter cover how modern hardware—including more complex processors such as the Intel Pentium series—is usually implemented. The basic principles of finite state control are explained, and different methods of implementation, including microprogramming, are examined. For the reader interested in understanding the processor and its performance in more depth, Sections 5.4 and 5.5 will be useful. For readers with an interest in modern hardware design,  [Section 5.7](#) covers microprogramming, a technique used to implement more complex control such as that present in IA-32 processors, and  [Section 5.8](#) describes how hardware design languages and CAD tools are used to implement hardware.

A Basic MIPS Implementation

We will be examining an implementation that includes a subset of the core MIPS instruction set:

- The memory-reference instructions load word (`lw`) and store word (`sw`)
- The arithmetic-logical instructions `add`, `sub`, `and`, `or`, and `sllt`
- The instructions branch equal (`beq`) and jump (`j`), which we add last

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions. However, the key principles used in creating a datapath and designing the control will be illustrated. The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the machine. Many of the key design principles introduced in Chapter 4 can be illustrated by looking at the implementation, such as the guidelines *Make the common case fast* and *Simplicity favors regularity*. In addition, most concepts used to implement the MIPS subset in this chapter and the next are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors, which are used increasingly in products ranging from VCRs to automobiles.

An Overview of the Implementation

In Chapters 2 and 3, we looked at the core MIPS instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require that we read two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact opcode.

Even across different instruction classes there are some similarities. For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. As we can see, the simplicity and regularity of the instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to write data for a store or read data for a load. An arithmetic-logical instruction must write the data from the ALU back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise the PC should be incremented by 4 to get the address of the next instruction.

Figure 5.1 shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

First, in several places, Figure 5.1 shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, and the data written into the register file can come from either the ALU or the data memory. In practice, these data lines cannot simply be wired together; we must add an element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. The multiplexor, which is described in detail in [Appendix B](#), selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

Second, several of the units must be controlled depending on the type of instruction. For example, the data memory must read on a load and write on a store. The register file must be written on a load and an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations, as we saw in Chapter 3. ([Appendix B](#) describes the detailed logic design of the ALU.) Like the muxes, these operations are directed by control lines that are set on the basis of various fields in the instruction.

Figure 5.2 shows the datapath of Figure 5.1 with the three required multiplexors added, as well as control lines for the major functional units. A control unit that has the instruction as an input is used to determine how to set the control lines for the functional units and two of the multiplexors. The third multiplexor, which determines whether $PC + 4$ or the branch destination address is written

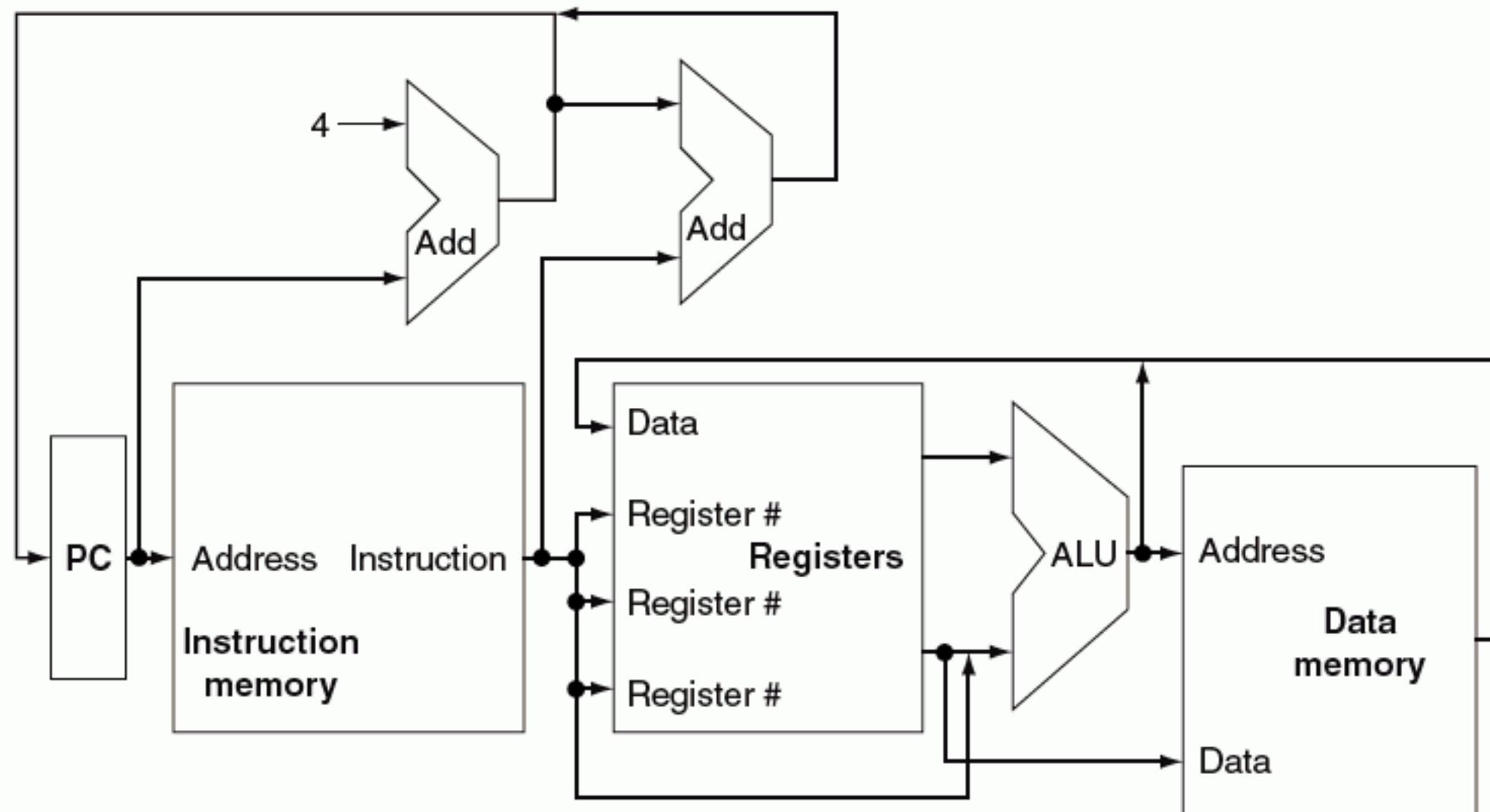


FIGURE 5.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes from either the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

into the PC, is set based on the zero output of the ALU, which is used to perform the comparison of a `beq` instruction. The regularity and simplicity of the MIPS instruction set means that a simple decoding process can be used to determine how to set the control lines.

In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, add a control unit to control what actions are taken for different instruction classes. Sections 5.3 and 5.4 describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of Figures 5.1 and 5.2. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

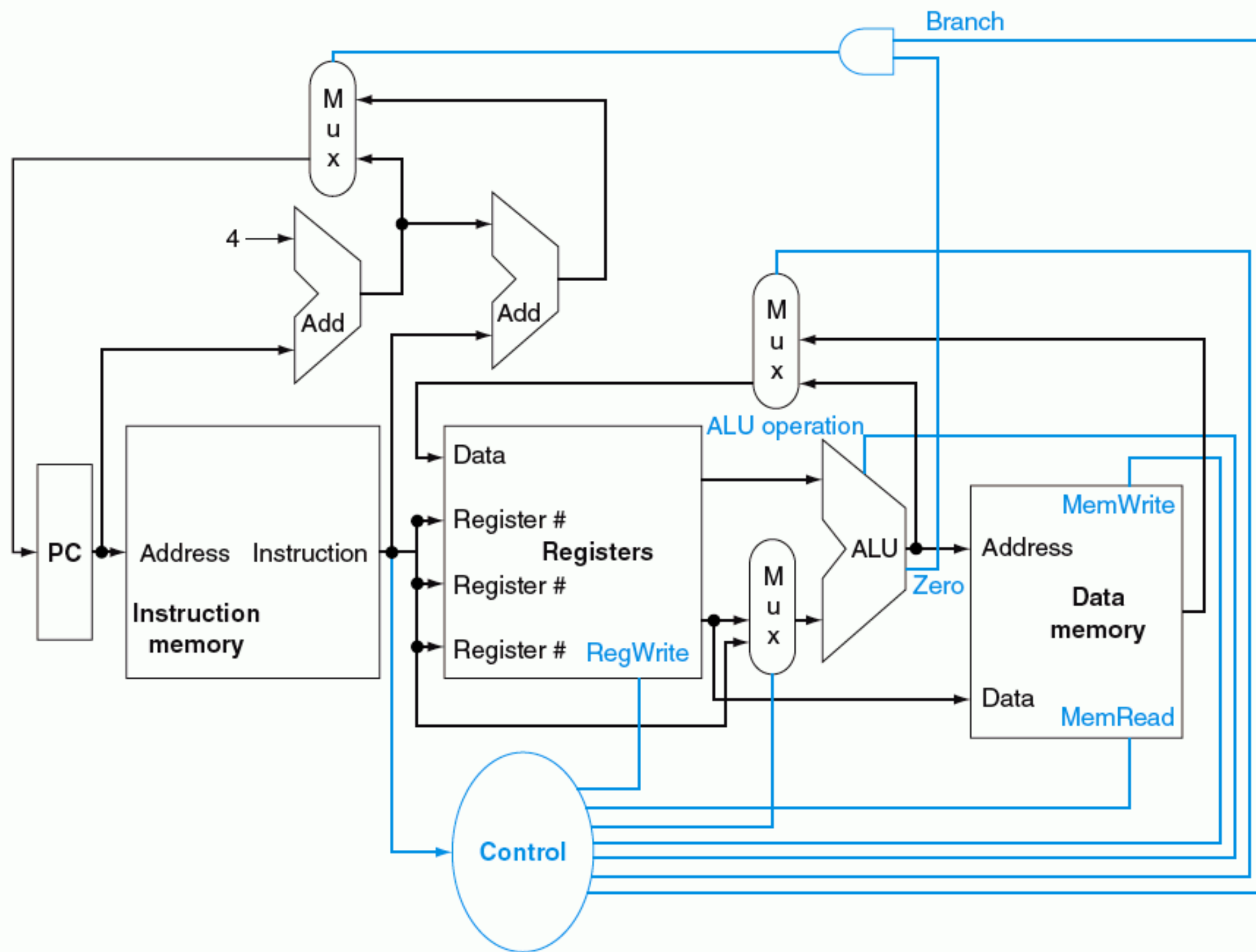


FIGURE 5.2 The basic implementation of the MIPS subset including the necessary multiplexors and control lines. The top multiplexor controls what value replaces the PC ($PC + 4$ or the branch destination address); the multiplexor is controlled by the gate that “ands” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The multiplexor whose output returns to the register file is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for a nonimmediate arithmetic-logical instruction) or from the offset field of the instruction (for an immediate operation, a load or store, or a branch). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This multicycle design is used

when we discuss more advanced control concepts, handling exceptions, and the use of hardware design languages in Sections 5.5 through 5.8.

The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories because

1. the format of data and instructions is different in MIPS and hence different memories are needed
2. having separate memories is less expensive
3. the processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle

Check Yourself

5.2 Logic Design Conventions

To discuss the design of a machine, we must decide how the logic implementing the machine will operate and how the machine is clocked. This section reviews a few key ideas in digital logic that we will use extensively in this chapter. If you have little or no background in digital logic, you will find it helpful to read through Appendix B before continuing.


The functional units in the MIPS implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all *combinational*, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU shown in Figure 5.1 and discussed in Chapter 3 and [Appendix B](#) is a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements **state elements** because, if we pulled the plug on the machine, we could restart it by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the machine had never lost power. Thus, these state elements completely characterize the machine. In Figure 5.1, the instruction and data memories as well as the registers are all examples of state elements.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines

state element A memory element.

when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see Appendix B), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our MIPS implementation also uses two other types of state elements: memories and registers, both of which appear in Figure 5.1. The clock is used to determine when the state element should be written; a state element can be read at any time.

Logic components that contain state are also called *sequential* because their outputs depend on both their inputs and the contents of the internal state. For example, the output from the functional unit representing the registers depends both on the register numbers supplied and on what was written into the registers previously. The operation of both the combinational and sequential elements and their construction are discussed in more detail in  Appendix B.

We will use the word *asserted* to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high, and *deassert* or *deasserted* to represent logical low.

Clocking Methodology

A **clocking methodology** defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes because, if a signal is written at the same time it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two! Needless to say, computer designs cannot tolerate such unpredictability. A clocking methodology is designed to prevent this circumstance.

For simplicity, we will assume an **edge-triggered** clocking methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge. Because only state elements can store a data value, any collection of combinational logic must have its inputs coming from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

Figure 5.3 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

For simplicity, we do not show a write **control signal** when a state element is written on every active clock edge. In contrast, if a state element is not updated on every clock, then an explicit write control signal is required. Both the clock signal and the write control signal are inputs, and the state element is changed only when the write control signal is asserted and a clock edge occurs.

clocking methodology The approach used to determine when data is valid and stable relative to the clock.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.

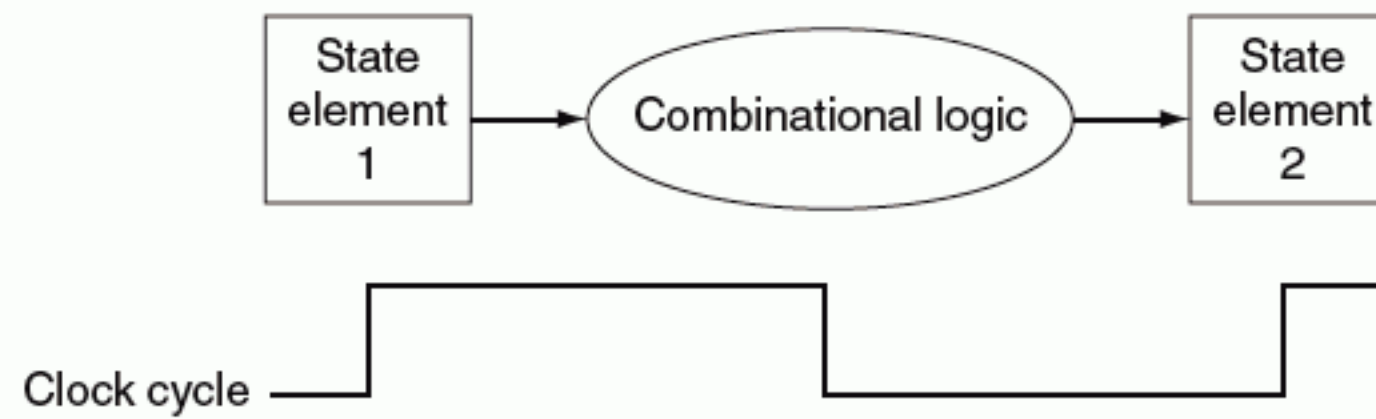


FIGURE 5.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements, including memory, are assumed to be edge-triggered.

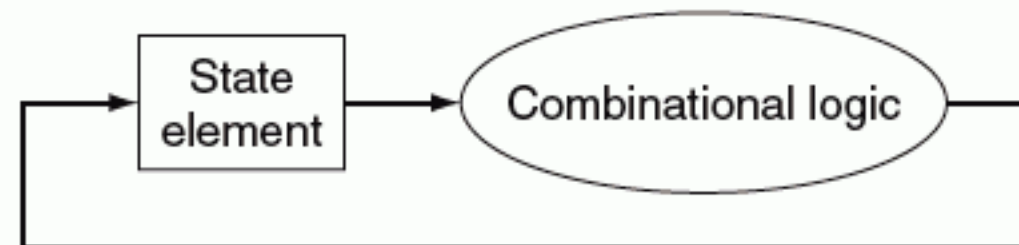


FIGURE 5.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within 1 clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and structures like the one shown in this figure.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle, as shown in Figure 5.4. It doesn't matter whether we assume that all writes take place on the rising clock edge or on the falling clock edge, since the inputs to the combinational logic block cannot change except on the chosen clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in Figure 5.4 works correctly. In [Appendix B](#) we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.

Nearly all of these state and logic elements will have inputs and outputs that are 32 bits wide, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 32 bits in width. The figures will indicate *buses*, which are signals wider than 1 bit, with thicker lines. At times we will want to combine several buses to form a wider bus; for example, we may want to obtain a 32-bit bus by combining two 16-bit

buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, **color** indicates a control signal as opposed to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

Check Yourself

True or false: Because the register file is both read and written on the same clock cycle, any MIPS datapath using edge-triggered writes must have more than one copy of the register file.

5.3

Building a Datapath

datapath element A functional unit used to operate on or hold data within a processor. In the MIPS implementation the datapath elements include the instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders.

program counter (PC) The register containing the address of the instruction in the program being executed

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instruction. Let's start by looking at which **datapath elements** each instruction needs. When we show the datapath elements, we will also show their control signals.

Figure 5.5 shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 5.5 also shows a register, which we call the **program counter (PC)**, that is used to hold the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU we described in Chapter 3 and designed in detail in Appendix B, simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in Figure 5.5, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure 5.6 shows how the three elements from Figure 5.5 are combined to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

Now let's consider the R-format instructions (see Figure 2.7 on page 67). They all read two registers, perform an ALU operation on the contents of the registers, and write the result. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes `add`, `sub`, `and`, `or`, and `slt`, which were intro-

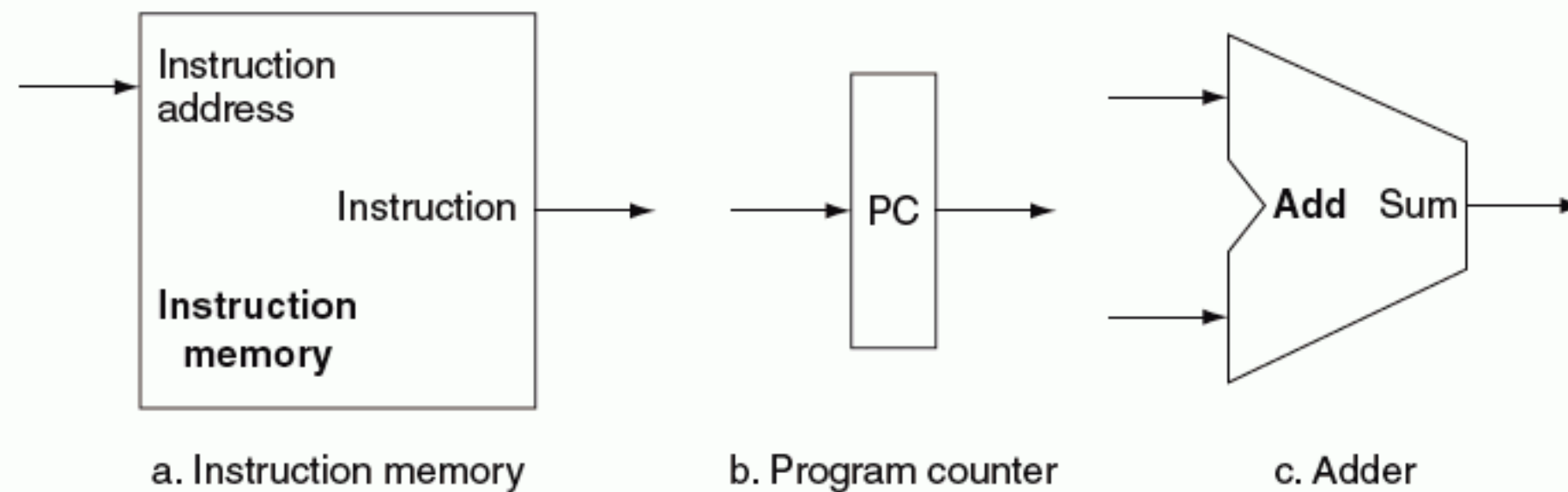


FIGURE 5.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that will be written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always perform an add of its two 32-bit inputs and place the result on its output.

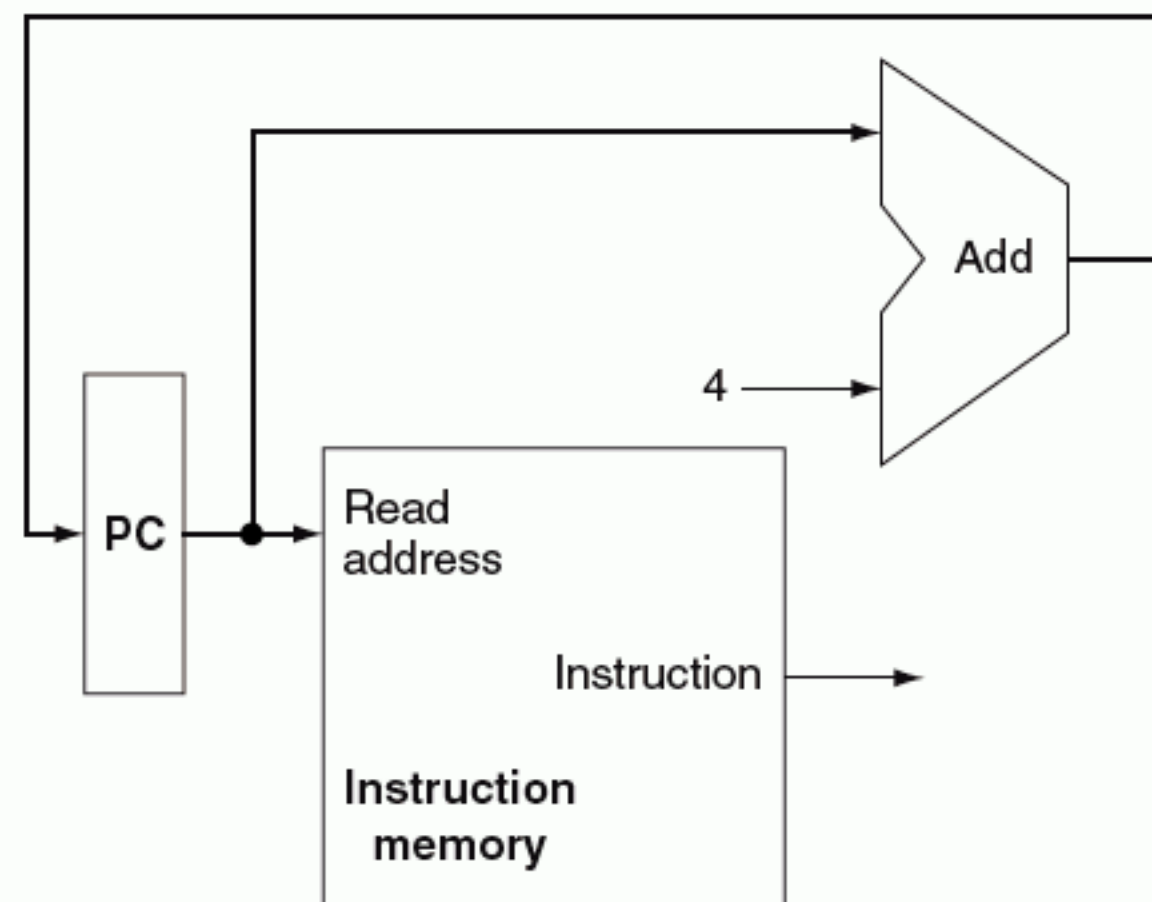


FIGURE 5.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

duced in Chapter 2. Recall that a typical instance of such an instruction is `add $t1,$t2,$t3`, which reads `$t2` and `$t3` and writes `$t1`.

The processor's 32 general-purpose registers are stored in a structure called a **register file**. A register file is a collection of registers in which any register can be

register file A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

read or written by specifying the number of the register in the file. The register file contains the register state of the machine. In addition, we will need an ALU to operate on the values read from the registers.

Because the R-format instructions have three register operands, we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the *register number* to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. Thus, we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data), as shown in Figure 5.7. The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

Figure 5.7 shows the ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The four-bit control signal of the ALU is described in detail in [Appendix B](#); we will review the ALU control shortly when we need to know how to set it.

Next, consider the MIPS load word and store word instructions, which have the general form `lw $t1,offset_value($t2)` or `sw $t1,offset_value($t2)`. These instructions compute a memory address by adding the base register, which is `$t2`, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in `$t1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `$t1`. Thus, we will need both the register file and the ALU from Figure 5.7.

In addition, we will need a unit to [sign-extend](#) the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, it has both read and write control signals, an address input, as well as an input for the data to be written into memory. Figure 5.8 shows these two elements.

The `beq` instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the [branch target address](#) relative to the branch instruction address. Its form is `beq $t1,$t2,offset`. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. There are two details in the definition of branch instructions (see Chapter 2) to which we must pay attention:

sign-extend To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

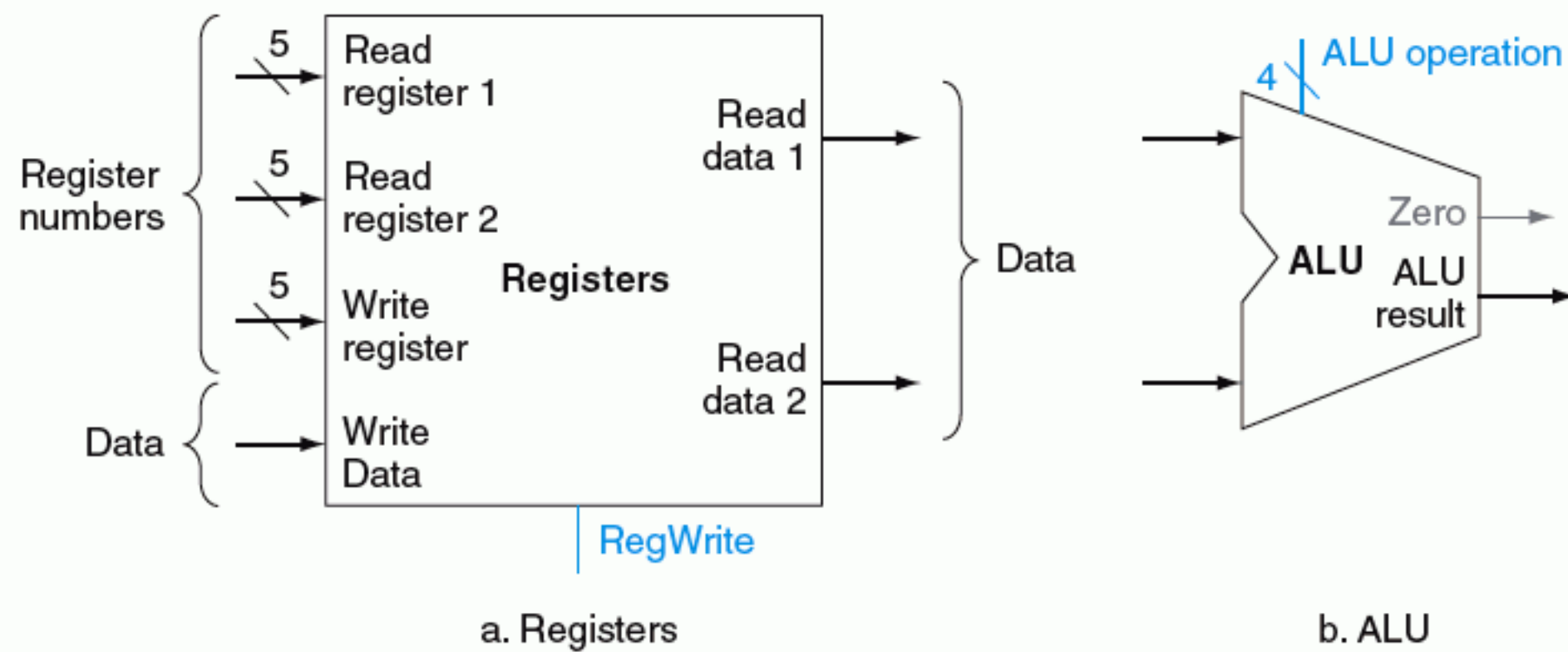


FIGURE 5.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of Appendix B. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in [Appendix B](#). We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 5.6, when we discuss exceptions; we omit it until then.

- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute $PC + 4$ (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of four.

To deal with the latter complication, we will need to shift the offset field by two.

In addition to computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch is taken**. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch is not taken**.

branch taken A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

branch not taken A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

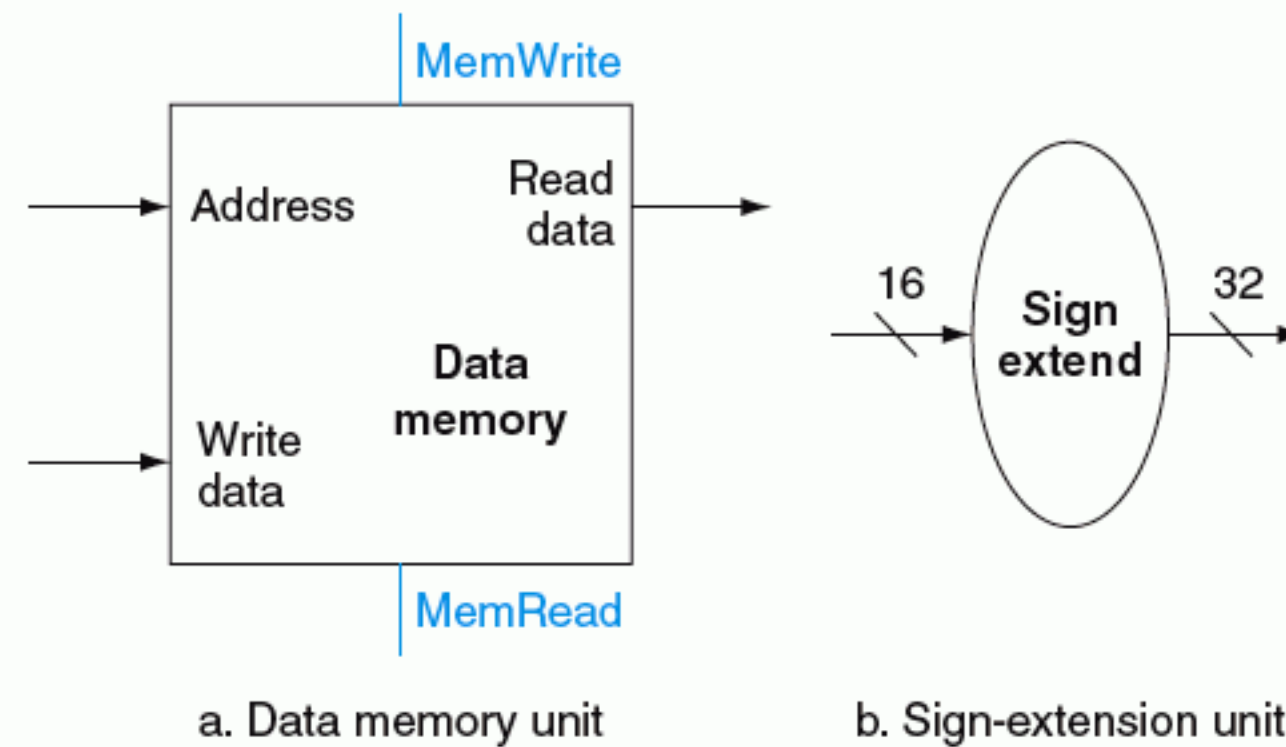


FIGURE 5.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 5.7, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 7. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 3). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section B.8 of [Appendix B](#) for a further discussion of how real memory chips work.

Thus, the branch datapath must do two operations: compute the branch target address and compare the register contents. (Branches also affect the instruction fetch portion of the datapath, as we will deal with shortly.) Because of the complexity of handling branches, we show the structure of the datapath segment that handles branches in Figure 5.9. To compute the branch target address, the branch datapath includes a sign extension unit, just like that in Figure 5.8, and an adder. To perform the compare, we need to use the register file shown in Figure 5.7 to supply the two register operands (although we will not need to write into the register file). In addition, the comparison can be done using the ALU we designed in Appendix B. Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits. This shift is accomplished simply by concatenating 00 to the jump offset, as described in Chapter 2.

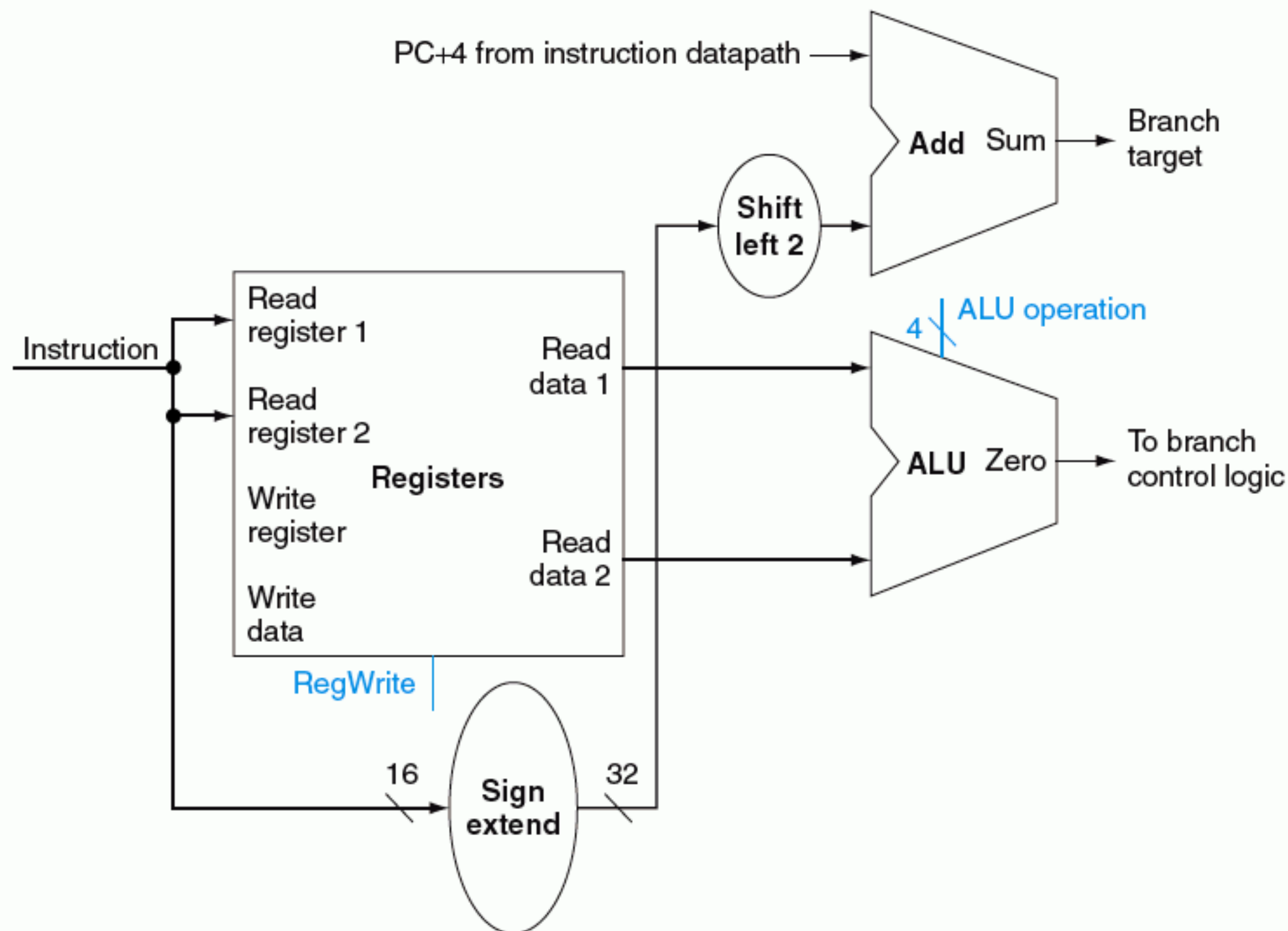


FIGURE 5.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the “shift” is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Elaboration: In the MIPS instruction set, branches are **delayed**, meaning that the instruction immediately following the branch is always executed, *independent* of whether the branch condition is true or false. When the condition is false, the execution looks like a normal branch. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address. The motivation for delayed branches arises from how pipelining affects branches (see Section 6.6). For simplicity, we ignore delayed branches in this chapter and implement a nondelayed `beq` instruction.

delayed branch A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

Creating a Single Datapath

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. The simplest datapath might attempt to execute all instructions in one clock cycle. This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

EXAMPLE

Building a Datapath

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 16-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

ANSWER

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. Figure 5.10 shows the operational portion of the combined datapath.

Now we can combine all the pieces to make a simple datapath for the MIPS architecture by adding the datapath for instruction fetch (Figure 5.6 on page 293),

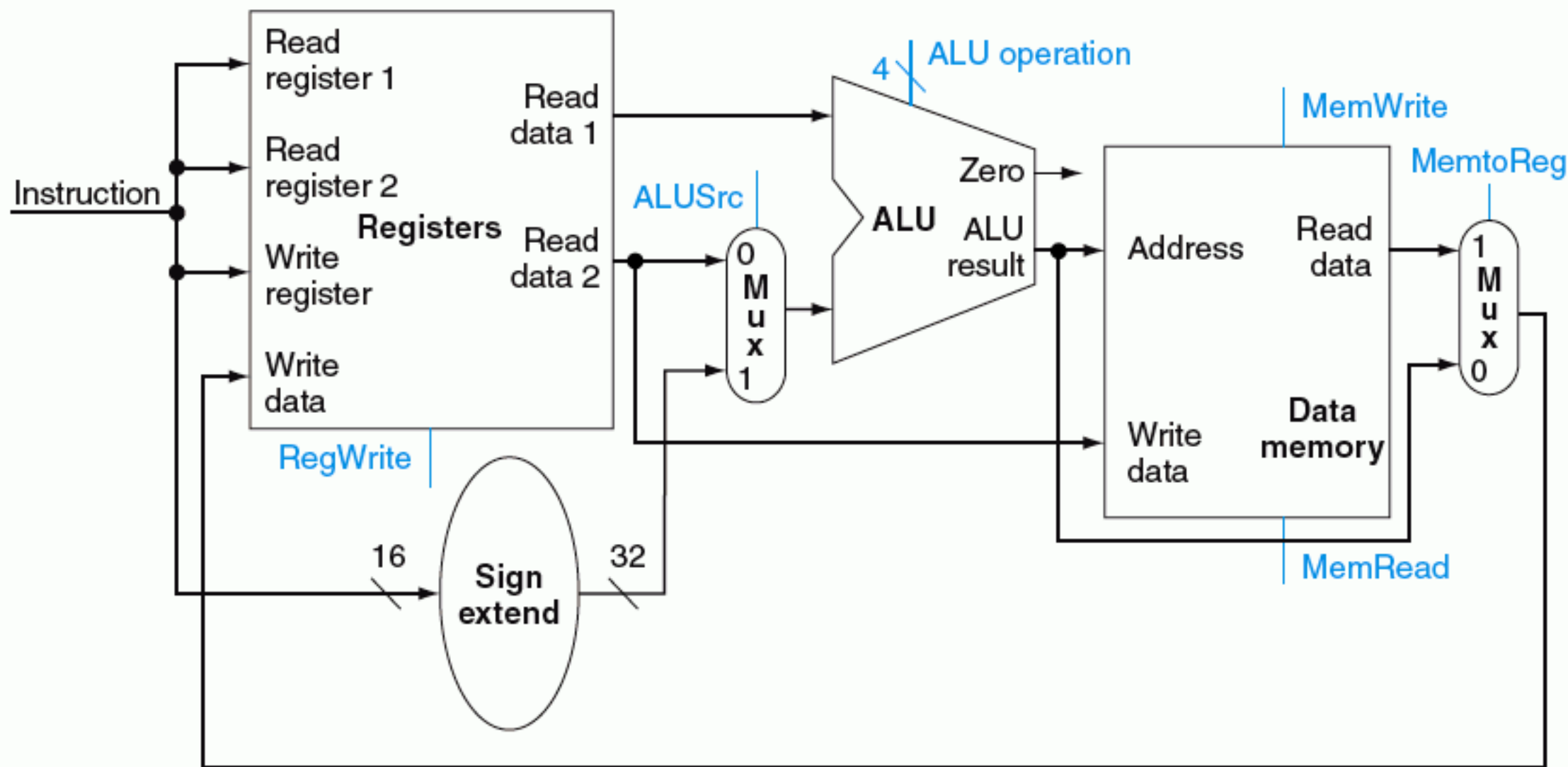


FIGURE 5.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 5.7 and 5.8 by adding multiplexors. Two multiplexors are needed, as described as in the example.

the datapath from R-type and memory instructions (Figure 5.10 on page 299), and the datapath for branches (Figure 5.9 on page 297). Figure 5.11 shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder in Figure 5.9 for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address ($PC + 4$) or the branch target address to be written into the PC.

Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

Which of the following is correct for a load instruction?

- MemtoReg should be set to cause the data from memory to be sent to the register file.
- MemtoReg should be set to cause the correct register destination to be sent to the register file.
- We do not care about the setting of MemtoReg.

**Check
Yourself**

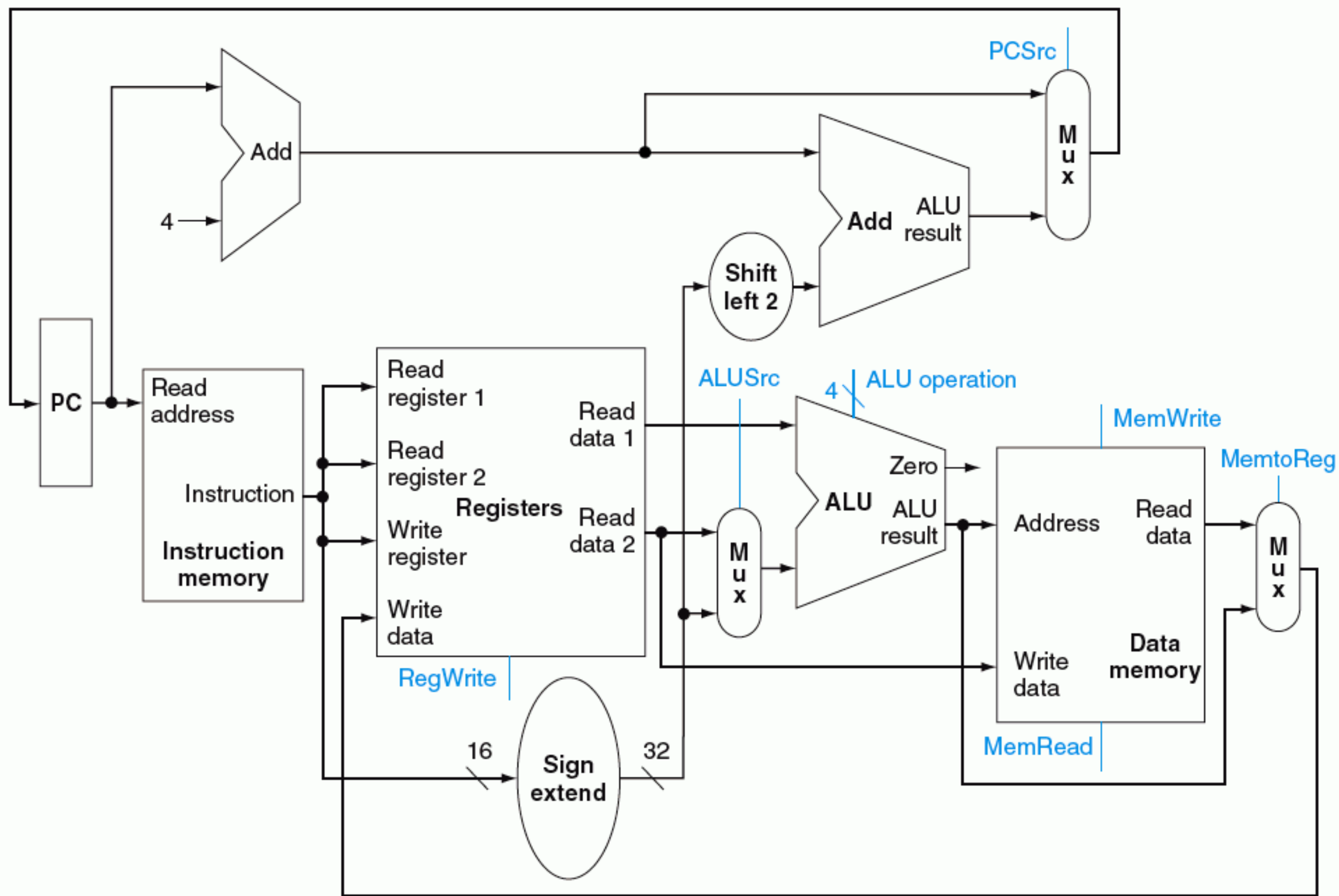


FIGURE 5.11 The simple datapath for the MIPS architecture combines the elements required by different instruction classes. This datapath can execute the basic instructions (load/store word, ALU operations, and branches) in a single clock cycle. An additional multiplexor is needed to integrate branches. The support for jumps will be added later.

5.4

A Simple Implementation Scheme

In this section, we look at what might be thought of as the simplest possible implementation of our MIPS subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, and, or, and set on less than. We will later enhance the design to include a jump instruction (j).

The ALU Control

As can be seen in Appendix B, the ALU has four control inputs. These bits were not encoded; hence, only 6 of the possible 16 possible input combinations are used in this subset. The MIPS ALU in [Appendix B](#) shows the 6 following combinations:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set.) For load word and store word instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction (see Chapter 2). For branch equal, the ALU must perform a subtraction.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously.

In Figure 5.12, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. For completeness, the relationship between the ALUOp bits and the instruction opcode is also shown. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the control unit is often performance-critical.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the three ALU operation control bits.

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 5.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don’t-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don’t-care terms and are replaced with XX in the truth table.

Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

As a step in designing this logic, it is useful to create a truth table for the interesting combinations of the function code field and the ALUOp bits, as we’ve done in Figure 5.13; this truth table shows how the 3-bit ALU control is set depending

on these two input fields. Since the full truth table is very large ($2^8 = 256$ entries) and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries that must be asserted and not showing those that are all zero or don't care. (This practice has a disadvantage, which we discuss in Section C.2 of Appendix C.)

Because in many instances we do not care about the values of some of the inputs and to keep the tables compact, we also include **don't-care terms**. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first line of the table in Figure 5.13, we always set the ALU control to 010, independent of the function code. In this case, then, the function code inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see Appendix B for more information.

Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical. Thus, rather than show the final steps here, we describe the process and the result in Section C.2 of Appendix C.

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in Figure 5.11 on page 300. To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load/store instructions. Figure 5.14 shows these formats.

There are several major observations about this instruction format that we will rely on:

- The op field, also called the **opcode**, is always contained in bits 31:26. We will refer to this field as Op[5:0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and for store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).

don't-care term An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

opcode The field that denotes the operation and format of an instruction.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0
a. R-type instruction						
Field	35 or 43	rs	rt	address		
Bit positions	31:26	25:21	20:16	15:0		
b. Load or store instruction						
Field	4	rs	rt	address		
Bit positions	31:26	25:21	20:16	15:0		
c. Branch instruction						

FIGURE 5.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, and, or, and s l t. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.

- The 16-bit offset for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

Using this information, we can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath. Figure 5.15 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

Figure 5.15 shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. Figure 5.16 describes the function of these seven control lines.

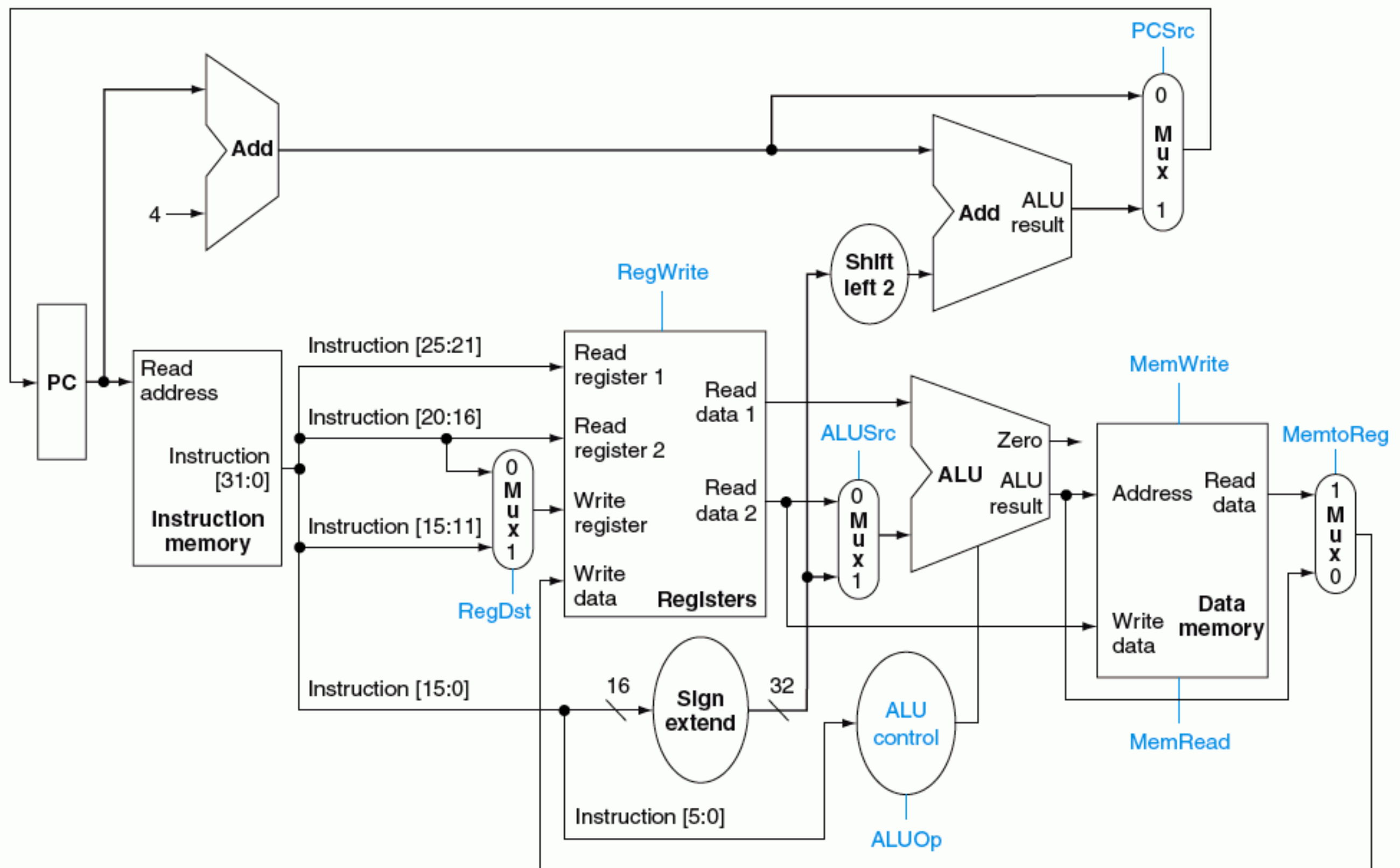


FIGURE 5.15 The datapath of Figure 5.12 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Now that we have looked at the function of each of the control signals, we can look at how to set them. The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception. That control line should be set if the instruction is branch on equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for equality comparison, is true. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

These nine control signals (seven from Figure 5.16 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits. Figure 5.17 shows the datapath with the control unit and the control signals.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 5.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. The clock is never gated externally to a state element, since this can create timing problems. (See [Appendix B](#) for further discussion of this problem.)

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X), for each of the opcode values. Figure 5.18 defines how the control signals should be set for each opcode; this information follows directly from Figures 5.12, 5.16, and 5.17.

Operation of the Datapath

With the information contained in Figures 5.16 and 5.18, we can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

Figure 5.19 shows the operation of the datapath for an R-type instruction, such as `add $t1, $t2, $t3`. Although everything occurs in 1 clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

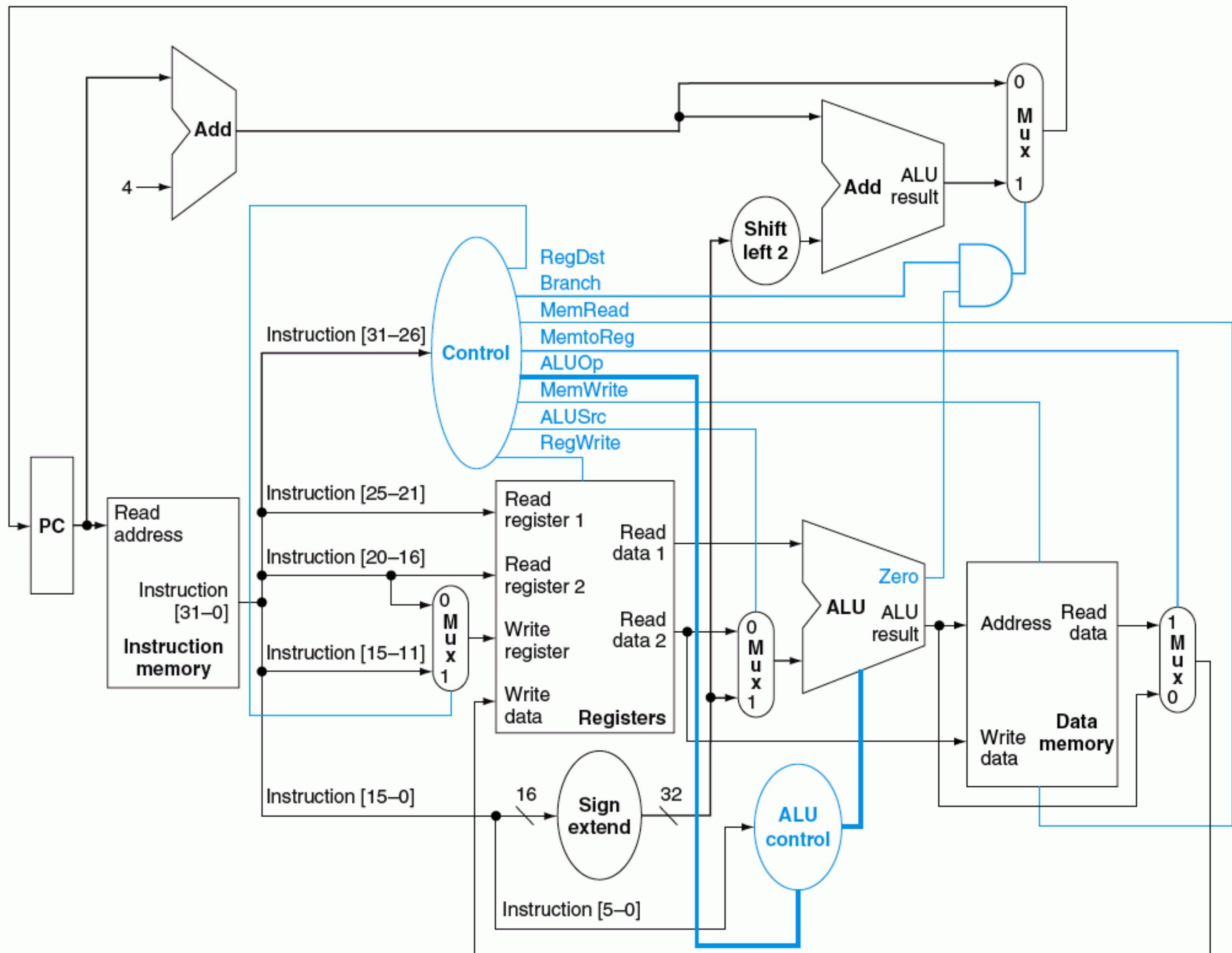


FIGURE 5.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus we drop the signal name in subsequent figures.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

FIGURE 5.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (add, sub, and, or, and slt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

1. The instruction is fetched, and the PC is incremented.
2. Two registers, \$t2 and \$t3, are read from the register file, and the main control unit computes the setting of the control lines during this step also.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register (\$t1).

Similarly, we can illustrate the execution of a load word, such as

```
lw $t1, offset($t2)
```

in a style similar to Figure 5.19. Figure 5.20 on page 310 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to the R-type executed in four):

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (\$t2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.

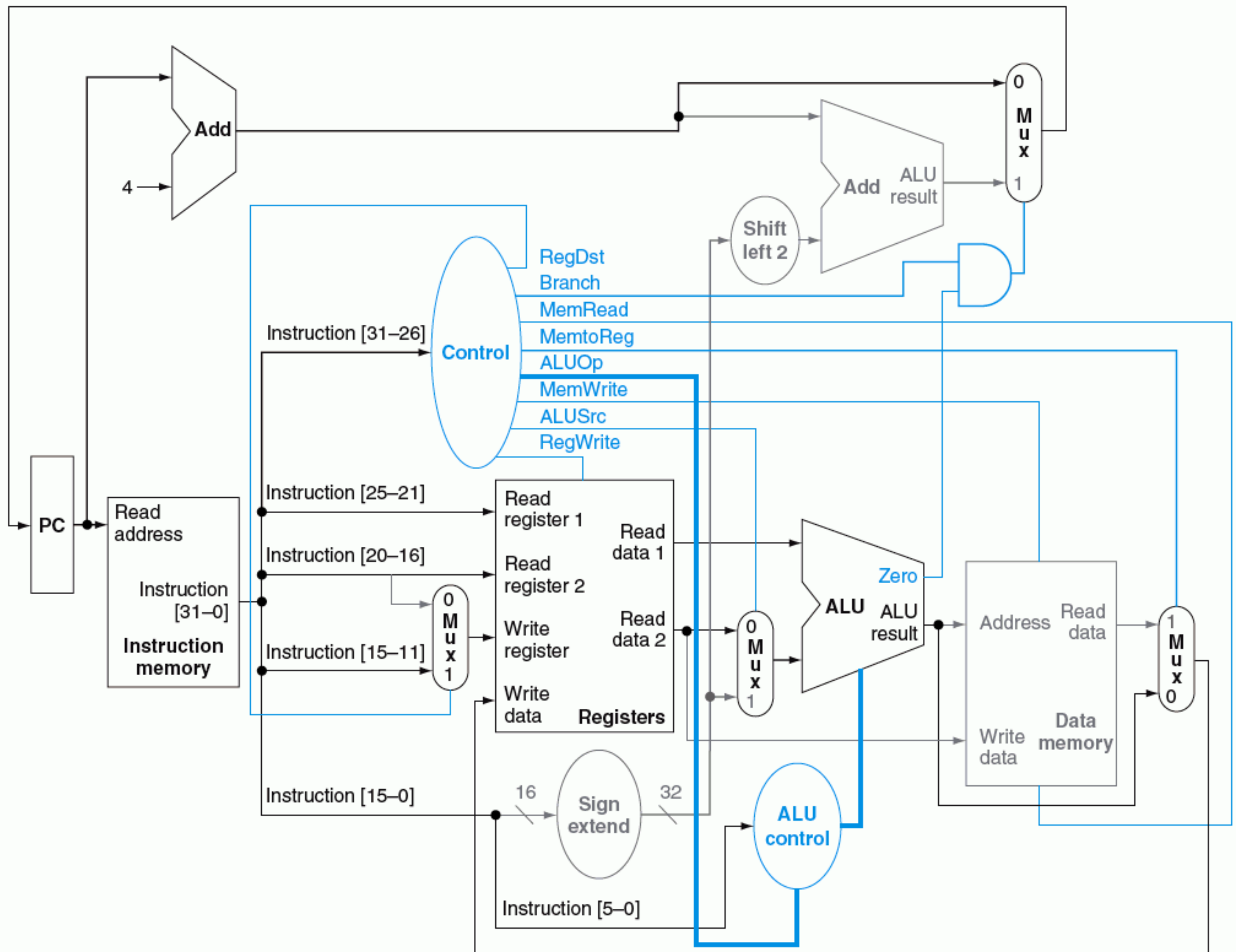


FIGURE 5.19 The datapath in operation for an R-type instruction such as `add $t1, $t2, $t3`. The control lines, datapath units, and connections that are active are highlighted.

5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction (`$t1`).

Finally, we can show the operation of the branch-on-equal instruction, such as `beq $t1, $t2, offset`, in the same fashion. It operates much like an R-format

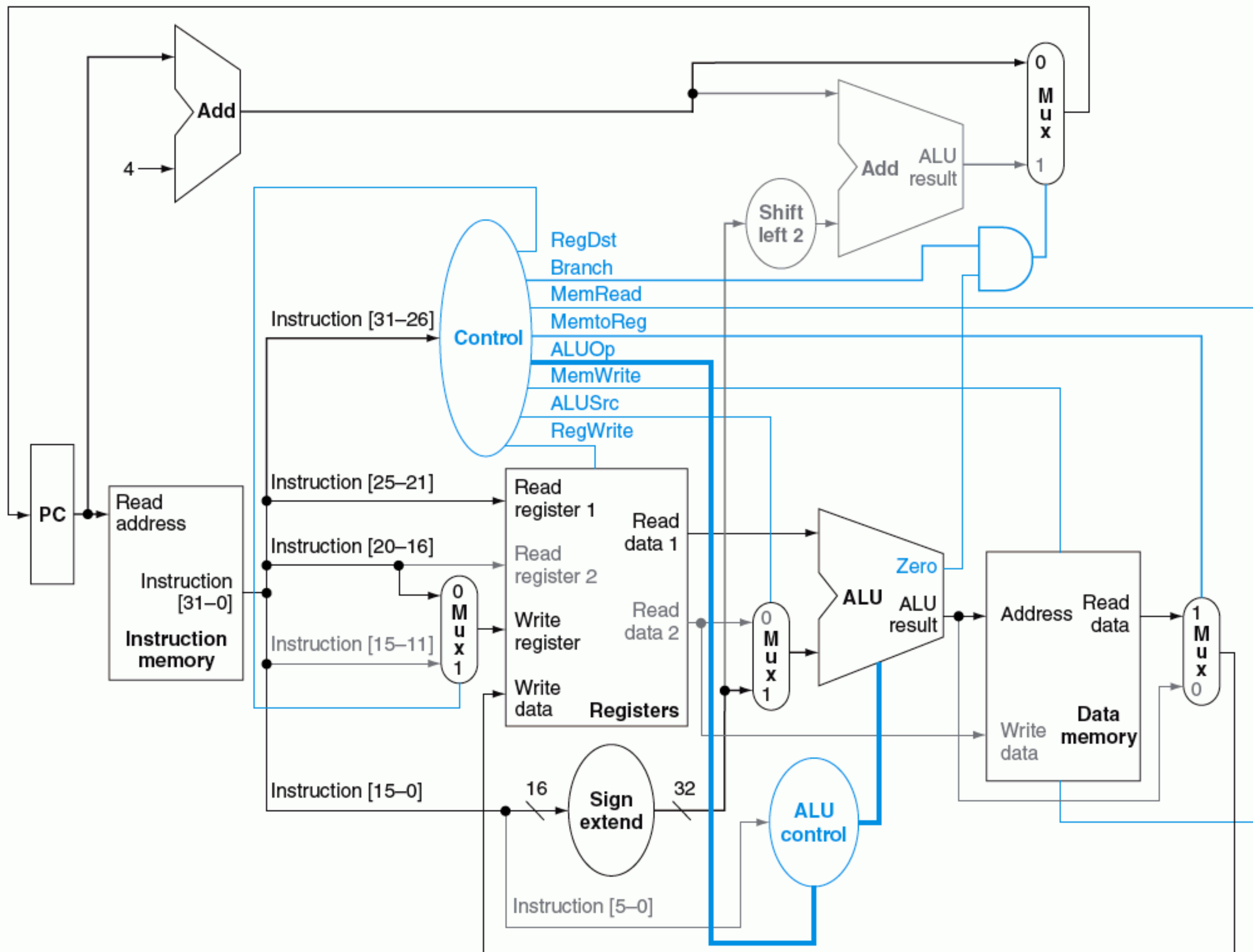


FIGURE 5.20 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

instruction, but the ALU output is used to determine whether the PC is written with $PC + 4$ or the branch target address. Figure 5.21 shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, $\$t1$ and $\$t2$, are read from the register file.

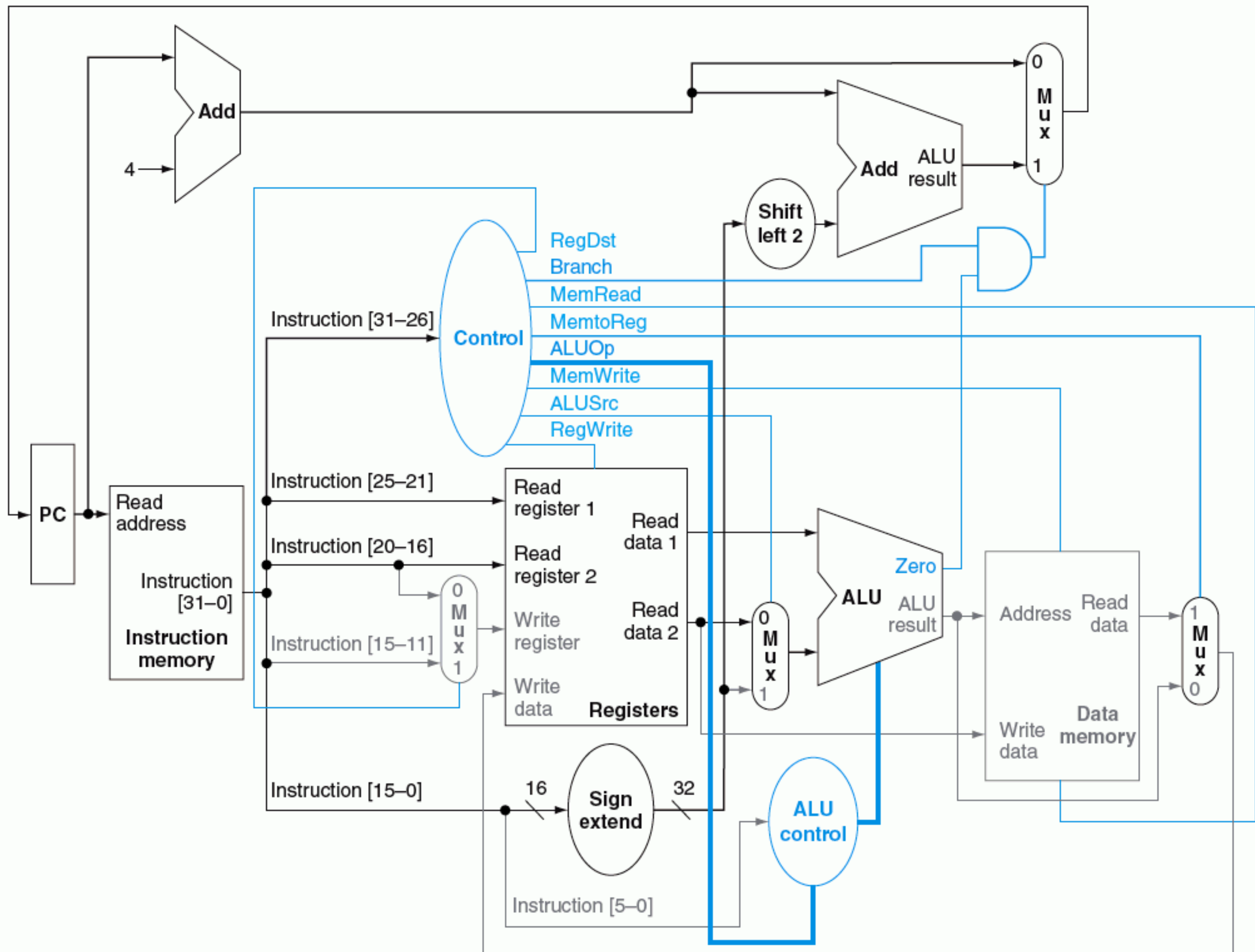


FIGURE 5.21 The datapath in operation for a branch equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

3. The ALU performs a subtract on the data values read from the register file. The value of $PC + 4$ is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

In the next section, we will examine machines that are truly sequential, namely, those in which each of these steps is a distinct clock cycle.

Finalizing the Control

Now that we have seen how the instructions operate in steps, let’s continue with the control implementation. The control function can be precisely defined using the contents of Figure 5.18 on page 308. The outputs are the control lines, and the input is the 6-bit opcode field, Op [5:0]. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

Figure 5.22 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in Section C.2 in [Appendix C](#).

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

single-cycle implementation Also called single clock cycle implementation. An implementation in which an instruction is executed in one clock cycle.

FIGURE 5.22 The control function for the simple **single-cycle implementation** is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don’t cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{\text{Op5}} \cdot \overline{\text{Op2}}$, since this is sufficient to distinguish the R-format instructions from lw, sw, and beq. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

Now, let’s add the jump instruction to show how the basic datapath and control can be extended to handle other instructions in the instruction set.

Implementing Jumps

Figure 5.17 on page 307 shows the implementation of many of the instructions we looked at in Chapter 2. One class of instructions missing is that of the jump instruction. Extend the datapath and control of Figure 5.17 to include the jump instruction. Describe how to set any new control lines.

EXAMPLE

The jump instruction looks somewhat like a branch instruction but computes the target PC differently and is not conditional. Like a branch, the low-order 2 bits of a jump address are always 00_{two}. The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction, as shown in Figure 5.23. The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4. Thus, we can implement a jump by storing into the PC the concatenation of

ANSWER

- the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
- the 26-bit immediate field of the jump instruction
- the bits 00_{two}

Figure 5.24 shows the addition of the control for jump added to Figure 5.17. An additional multiplexor is used to select the source for the new PC value, which is either the incremented PC (PC + 4), the branch target PC, or the jump target PC. One additional control signal is needed for the additional multiplexor. This control signal, called *Jump*, is asserted only when the instruction is a jump—that is, when the opcode is 2.

Field	000010	address
Bit positions	31:26	25:0

FIGURE 5.23 Instruction format for the jump instruction (opcode = 2). The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

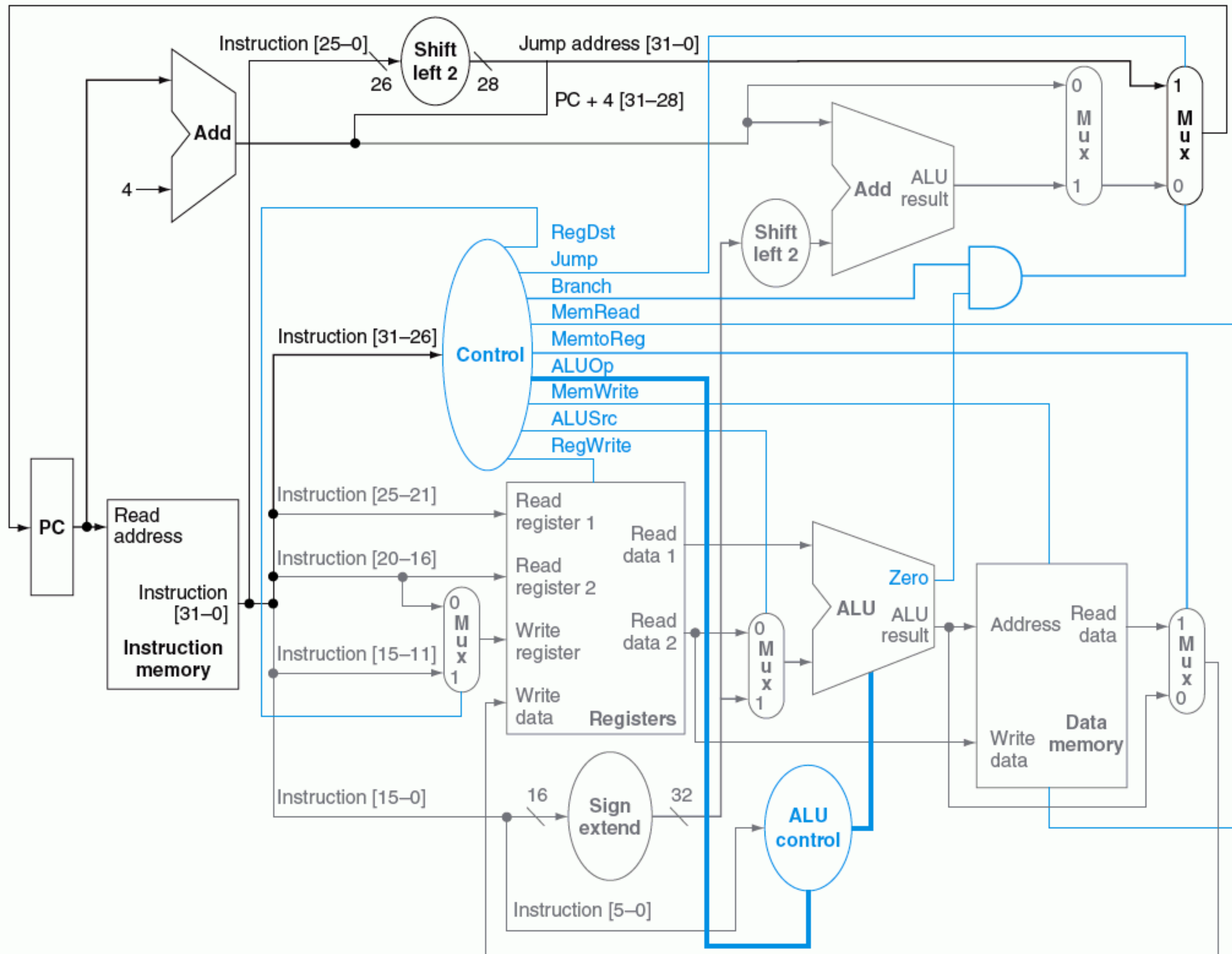


FIGURE 5.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Why a Single-Cycle Implementation Is Not Used Today

Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient. To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design, and the CPI

(see Chapter 4) will therefore be 1. Of course, the clock cycle is determined by the longest possible path in the machine. This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1, the overall performance of a single-cycle implementation is not likely to be very good, since several of the instruction classes could fit in a shorter clock cycle.

Performance of Single-Cycle Machines

Assume that the operation times for the major functional units in this implementation are the following:

- Memory units: 200 picoseconds (ps)
- ALU and adders: 100 ps
- Register file (read or write): 50 ps

Assuming that the multiplexors, control unit, PC accesses, sign extension unit, and wires have no delay, which of the following implementations would be faster and by how much?

1. An implementation in which every instruction operates in 1 clock cycle of a fixed length.
2. An implementation where every instruction executes in 1 clock cycle using a variable-length clock, which for each instruction is only as long as it needs to be. (Such an approach is not terribly practical, but it will allow us to see what is being sacrificed when all the instructions must execute in a single clock of the same length.)

To compare the performance, assume the following instruction mix: 25% loads, 10% stores, 45% ALU instructions, 15% branches, and 5% jumps.

Let's start by comparing the CPU execution times. Recall from Chapter 4 that

$$\text{CPU execution time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

Since CPI must be 1, we can simplify this to

$$\text{CPU execution time} = \text{Instruction count} \times \text{Clock cycle time}$$

EXAMPLE

ANSWER

We need only find the clock cycle time for the two implementations, since the instruction count and CPI are the same for both implementations. The critical path for the different instruction classes is as follows:

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Using these critical paths, we can compute the required length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

The clock cycle for a machine with a single clock for all instructions will be determined by the longest instruction, which is 600 ps. (This timing is approximate, since our timing model is quite simplistic. In reality, the timing of modern digital systems is complex.)

A machine with a variable clock will have a clock cycle that varies between 200 ps and 600 ps. We can find the average clock cycle length for a machine with a variable-length clock using the information above and the instruction frequency distribution.


Thus, the average time per instruction with a variable clock is

$$\begin{aligned} \text{CPU clock cycle} &= 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% \\ &= 447.5 \text{ ps} \end{aligned}$$

Since the variable clock implementation has a shorter average clock cycle, it is clearly faster. Let's find the performance ratio:

$$\begin{aligned}
 \frac{\text{CPU performance}_{\text{variable clock}}}{\text{CPU performance}_{\text{single clock}}} &= \frac{\text{CPU execution time}_{\text{single clock}}}{\text{CPU execution time}_{\text{variable clock}}} \\
 &= \left(\frac{\text{IC} \times \text{CPU clock cycle}_{\text{single clock}}}{\text{IC} \times \text{CPU clock cycle}_{\text{variable clock}}} = \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}} \right) \\
 &= \frac{600}{447.5} = 1.34
 \end{aligned}$$

The variable clock implementation would be 1.34 times faster. Unfortunately, implementing a variable-speed clock for each instruction class is extremely difficult, and the overhead for such an approach could be larger than any advantage gained. As we will see in the next section, an alternative is to use a shorter clock cycle that does less work and then vary the number of clock cycles for the different instruction classes.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early machines with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all. An example of this is shown in the  [For More Practice Exercise 5.4](#).

Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, we can't use implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates our key design principle of making the common case fast. In addition, in this single-cycle implementation, each functional unit can be used only once per clock; therefore, some functional units must be duplicated, raising the cost of the implementation. A single-cycle design is inefficient both in its performance and in its hardware cost!

We can avoid these difficulties by using implementation techniques that have a shorter clock cycle—derived from the basic functional unit delays—and that require multiple clock cycles for each instruction. The next section explores this alternative implementation scheme. In Chapter 6, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath, but is much more efficient. Pipelining gains efficiency by overlapping the execution of multiple instructions, increasing hardware utilization and improving performance. For those readers interested primarily in the high-level concepts used in processors, the material of this section is sufficient to read the introductory sections of Chapter 6 and understand the basic functional-

ity of a pipelined processor. For those, who want to understand how the hardware really implements the control, forge ahead!

Check Yourself

Look at the control signal in Figure 5.22 on page 312. Can any control signal in the figure be replaced by the inverse of another? (Hint: Take into account the don't cares.) If so, can you use one signal for the other without adding an inverter?

5.5

A Multicycle Implementation

multicycle implementation Also called multiple clock cycle implementation. An implementation in which an instruction is executed in multiple clock cycles.

In an earlier example, we broke each instruction into a series of steps corresponding to the functional unit operations that were needed. We can use these steps to create a **multicycle implementation**. In a multicycle implementation, each *step* in the execution will take 1 clock cycle. The multicycle implementation allows a functional unit to be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycles and the ability to share functional units within the execution of a single instruction are the major advantages of a multicycle design. Figure 5.25 shows the abstract version of the mul-

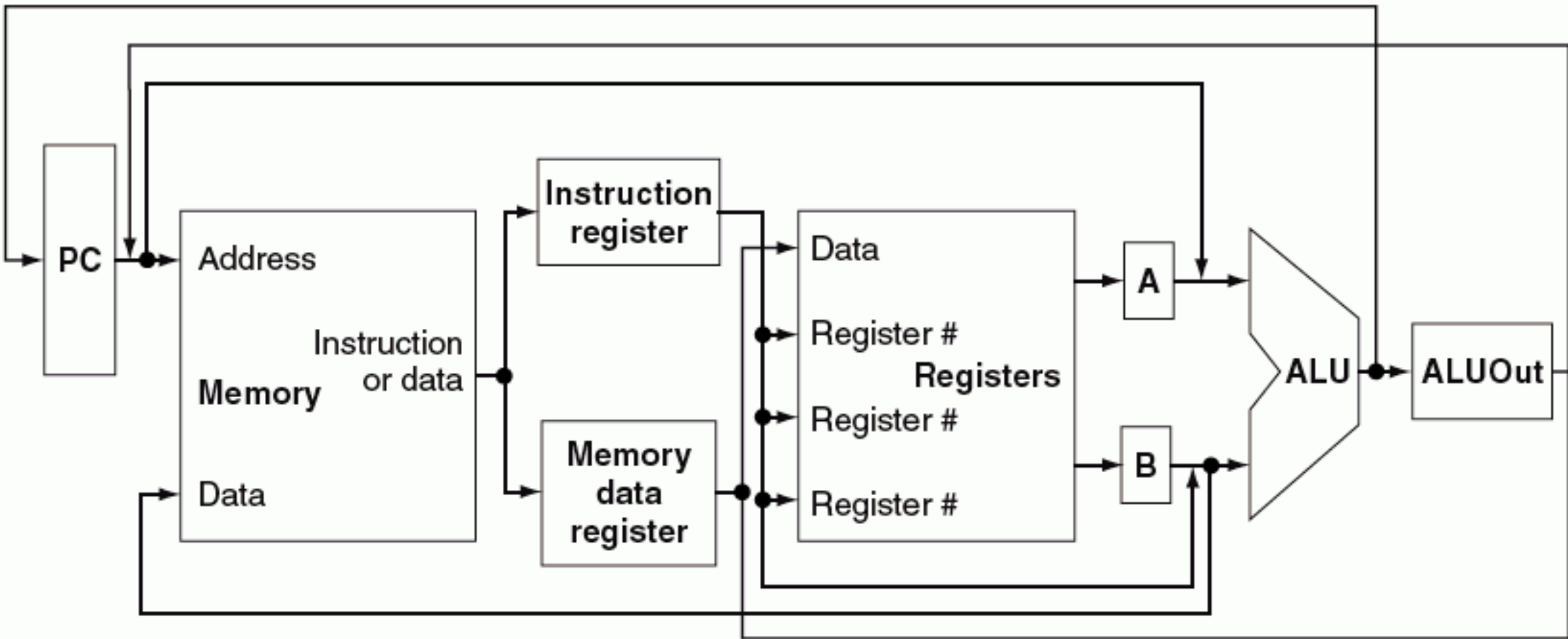


FIGURE 5.25 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexers as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

ticycle datapath. If we compare Figure 5.25 to the datapath for the single-cycle version in Figure 5.11 on page 300, we can see the following differences:

- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

At the end of a clock cycle, all data that is used in subsequent clock cycles must be stored in a state element. Data used by *subsequent instructions* in a later clock cycle is stored into one of the programmer-visible state elements: the register file, the PC, or the memory. In contrast, data used by the *same instruction* in a later cycle must be stored into one of these additional registers.

Thus, the position of the additional registers is determined by the two factors: what combinational units will fit in one clock cycle and what data are needed in later cycles implementing the instruction. In this multicycle design, we assume that the clock cycle can accommodate at most one of the following operations: a memory access, a register file access (two reads or one write), or an ALU operation. Hence, any data produced by one of these three functional units (the memory, the register file, or the ALU) must be saved, into a temporary register for use on a later cycle. If it were not saved then the possibility of a timing race could occur, leading to the use of an incorrect value.

The following temporary registers are added to meet these requirements:

- The Instruction register (IR) and the Memory data register (MDR) are added to save the output of the memory for an instruction read and a data read, respectively. Two separate registers are used, since, as will be clear shortly, both values are needed during the same clock cycle.
- The A and B registers are used to hold the register operand values read from the register file.
- The ALUOut register holds the output of the ALU.

All the registers except the IR hold data only between a pair of adjacent clock cycles and will thus not need a write control signal. The IR needs to hold the instruction until the end of execution of that instruction, and thus will require a write control signal. This distinction will become more clear when we show the individual clock cycles for each instruction.

Because several functional units are shared for different purposes, we need both to add multiplexors and to expand existing multiplexors. For example, since one memory is used for both instructions and data, we need a multiplexor to select between the two sources for a memory address, namely, the PC (for instruction access) and ALUOut (for data access).

Replacing the three ALUs of the single-cycle datapath by a single ALU means that the single ALU must accommodate all the inputs that used to go to the three different ALUs. Handling the additional inputs requires two changes to the datapath:

1. An additional multiplexor is added for the first ALU input. The multiplexor chooses between the A register and the PC.
2. The multiplexor on the second ALU input is changed from a two-way to a four-way multiplexor. The two additional inputs to the multiplexor are the constant 4 (used to increment the PC) and the sign-extended and shifted offset field (used in the branch address computation).

Figure 5.26 shows the details of the datapath with these additional multiplexors. By introducing a few registers and multiplexors, we are able to reduce the number of memory units from two to one and eliminate two adders. Since registers and multiplexors are fairly small compared to a memory unit or ALU, this could yield a substantial reduction in the hardware cost.

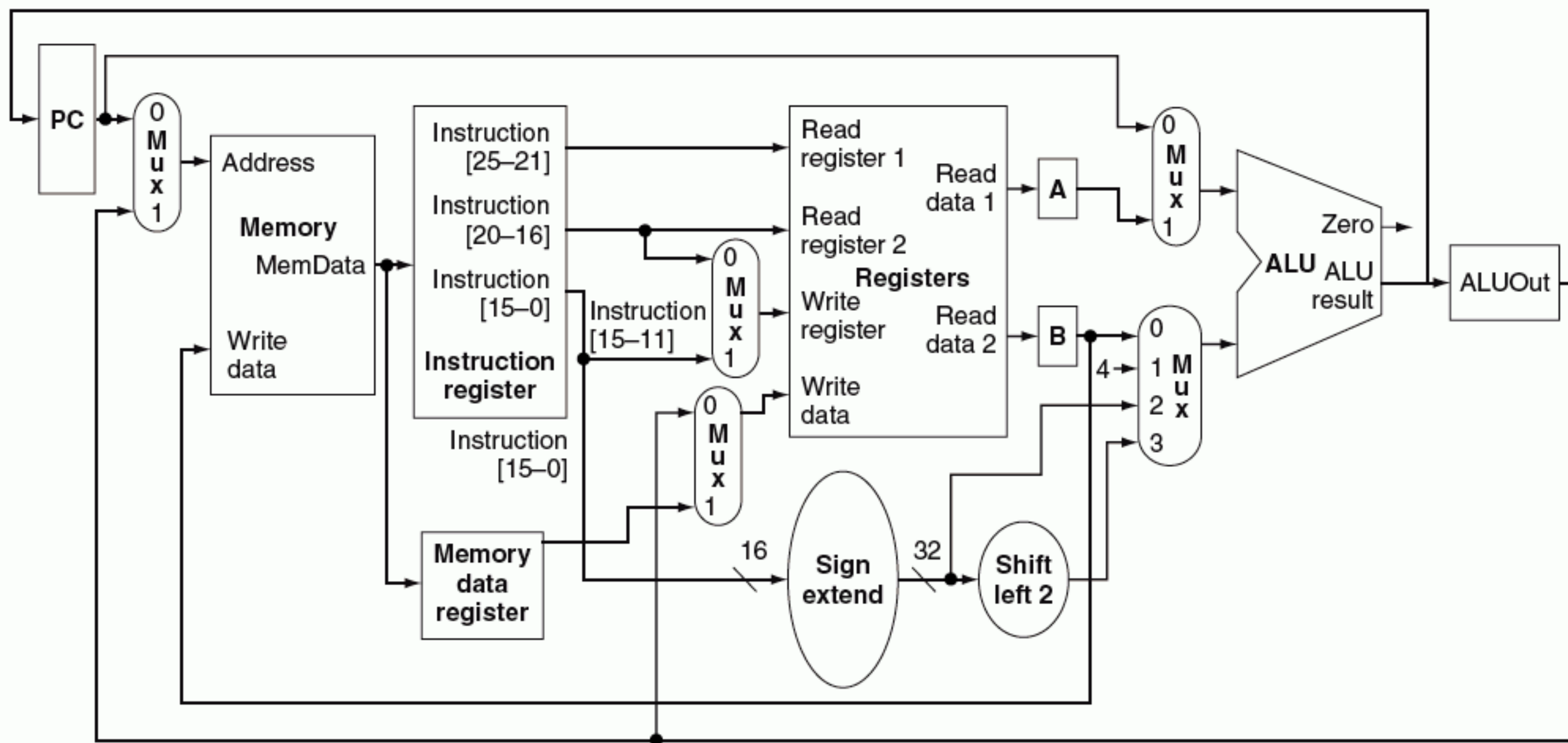


FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

Because the datapath shown in Figure 5.26 takes multiple clock cycles per instruction, it will require a different set of control signals. The programmer-visible state units (the PC, the memory, and the registers) as well as the IR will need write control signals. The memory will also need a read signal. We can use the ALU control unit from the single-cycle datapath (see Figure 5.13 and [Appendix C](#)) to control the ALU here as well. Finally, each of the two-input multiplexors requires a single control line, while the four-input multiplexor requires two control lines. Figure 5.27 shows the datapath of Figure 5.26 with these control lines added.

The multicycle datapath still requires additions to support branches and jumps; after these additions, we will see how the instructions are sequenced and then generate the datapath control.

With the jump instruction and branch instruction, there are three possible sources for the value to be written into the PC:

1. The output of the ALU, which is the value $PC + 4$ during instruction fetch. This value should be stored directly into the PC.
2. The register ALUOut, which is where we will store the address of the branch target after it is computed.
3. The lower 26 bits of the Instruction register (IR) shifted left by two and concatenated with the upper 4 bits of the incremented PC, which is the source when the instruction is a jump.

As we observed when we implemented the single-cycle control, the PC is written both unconditionally and conditionally. During a normal increment and for jumps, the PC is written unconditionally. If the instruction is a conditional branch, the incremented PC is replaced with the value in ALUOut only if the two designated registers are equal. Hence, our implementation uses two separate control signals: PCWrite, which causes an unconditional write of the PC, and PCWriteCond, which causes a write of the PC if the branch condition is also true.

We need to connect these two control signals to the PC write control. Just as we did in the single-cycle datapath, we will use a few gates to derive the PC write control signal from PCWrite, PCWriteCond, and the Zero signal of the ALU, which is used to detect if the two register operands of a beq are equal. To determine whether the PC should be written during a conditional branch, we AND together the Zero signal of the ALU with the PCWriteCond. The output of this AND gate is then ORed with PCWrite, which is the unconditional PC write signal. The output of this OR gate is connected to the write control signal for the PC.

Figure 5.28 shows the complete multicycle datapath and control unit, including the additional control signals and multiplexor for implementing the PC updating.

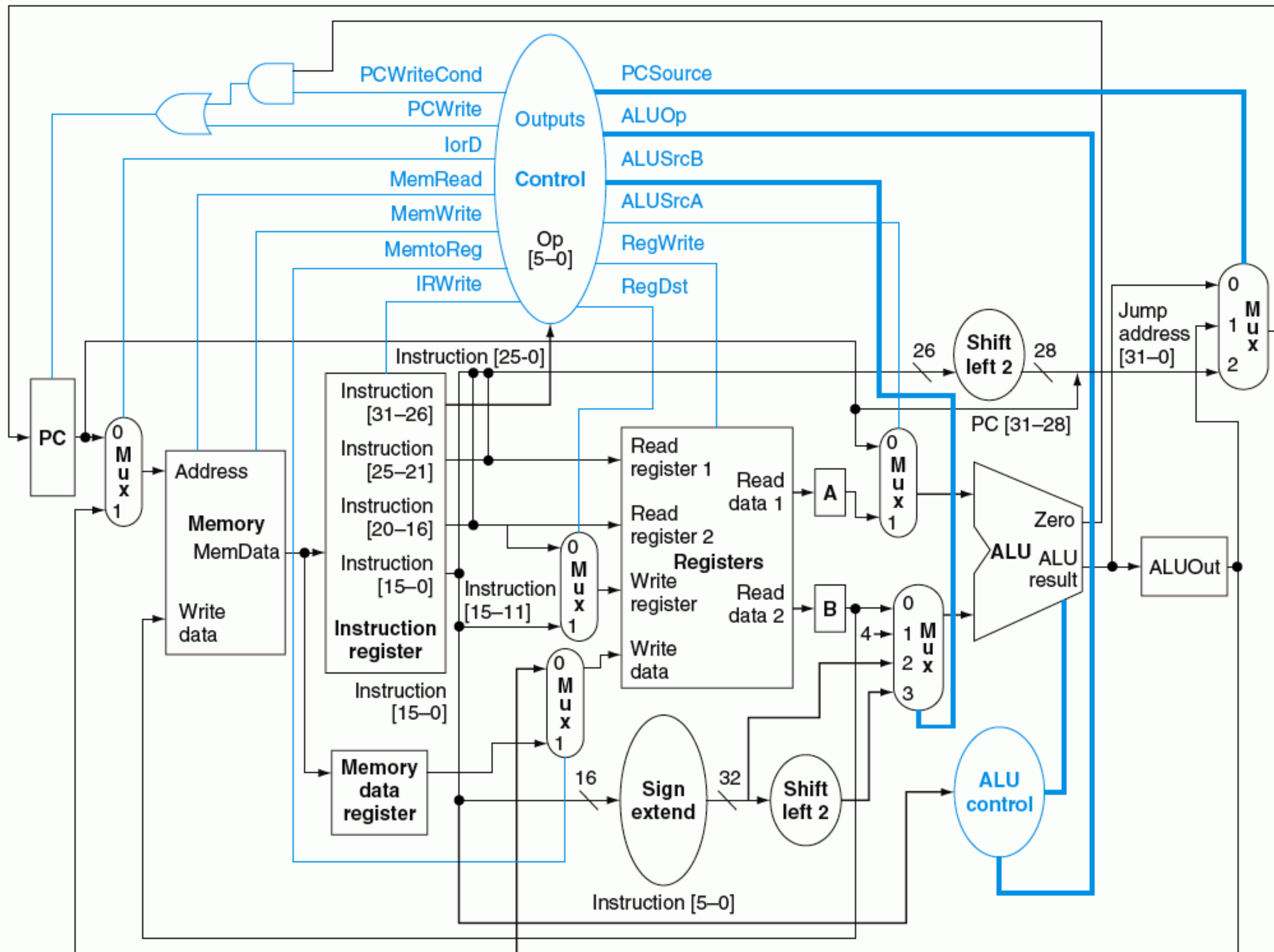


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

and multiple readers of the value. Just as we reduced the number of functional units for the datapath, we can reduce the number of buses interconnecting these units by sharing the buses. For example, there are six sources coming to the ALU; however, only two of them are needed at any one time. Thus, a pair of buses can be used to hold values that are being sent to the ALU. Rather than placing a large multiplexor in front of the

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSrc.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSrc	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

FIGURE 5.29 The action caused by the setting of each control signal in Figure 5.28 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSrc) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

ALU, a designer can use a shared bus and then ensure that only one of the sources is driving the bus at any point. Although this saves signal lines, the same number of control lines will be needed to control what goes on the bus. The major drawback to using such bus structures is a potential performance penalty, since a bus is unlikely to be as fast as a point-to-point connection.

Breaking the Instruction Execution into Clock Cycles

Given the datapath in Figure 5.28, we now need to look at what should happen in each clock cycle of the multicycle execution, since this will determine what additional control signals may be needed, as well as the setting of the control signals. Our goal in breaking the execution into clock cycles should be to maximize performance. We can begin by breaking the execution of any instruction into a series of steps, each taking one clock cycle, attempting to keep the amount of work per cycle roughly equal. For example, we will restrict each step to contain at most one ALU operation, or one register file access, or one memory access. With this restriction, the clock cycle could be as short as the longest of these operations.

Recall that at the end of every clock cycle any data values that will be needed on a subsequent cycle must be stored into a register, which can be either one of the major state elements (e.g., the PC, the register file, or the memory), a temporary register written on every clock cycle (e.g., A, B, MDR, or ALUOut), or a temporary register with write control (e.g., IR). Also remember that because our design is edge-triggered, we can continue to read the current value of a register; the new value does not appear until the next clock cycle.

In the single-cycle datapath, each instruction uses a set of datapath elements to carry out its execution. Many of the datapath elements operate in series, using the output of another element as an input. Some datapath elements operate in parallel; for example, the PC is incremented and the instruction is read at the same time. A similar situation exists in the multicycle datapath. All the operations listed in one step occur in parallel within 1 clock cycle, while successive steps operate in series in different clock cycles. The limitation of one ALU operation, one memory access, and one register file access determines what can fit in one step.

Notice that we distinguish between reading from or writing into the PC or one of the stand-alone registers and reading from or writing into the register file. In the former case, the read or write is part of a clock cycle, while reading or writing a result into the register file takes an additional clock cycle. The reason for this distinction is that the register file has additional control and access overhead compared to the single stand-alone registers. Thus, keeping the clock cycle short motivates dedicating separate clock cycles for register file accesses.

The potential execution steps and their actions are given below. Each MIPS instruction needs from three to five of these steps:

1. Instruction fetch step

Fetch the instruction from memory and compute the address of the next sequential instruction:

```
IR <= Memory[PC];  
PC <= PC + 4;
```


Operation: Send the PC to the memory as the address, perform a read, and write the instruction into the Instruction register (IR), where it will be stored. Also, increment the PC by 4. We use the symbol “<=” from Verilog; it indicates that all right-hand sides are evaluated and then all assignments are made, which is effectively how the hardware executes during the clock cycle.

To implement this step, we will need to assert the control signals MemRead and IRWrite, and set IorD to 0 to select the PC as the source of the address. We also increment the PC by 4, which requires setting the ALUSrcA signal to 0 (sending the PC to the ALU), the ALUSrcB signal to 01 (sending 4 to the ALU), and ALUOp to 00 (to make the ALU add). Finally, we will also want to store the incremented instruction address back into the PC, which requires setting PC source to 00 and setting PCWrite. The increment of the PC and the instruction memory access can occur in parallel. The new value of the PC is not visible until the next clock cycle. (The incremented PC will also be stored into ALUOut, but this action is benign.)

2. Instruction decode and register fetch step

In the previous step and in this one, we do not yet know what the instruction is, so we can perform only actions that are either applicable to all instructions (such as fetching the instruction in step 1) or are not harmful, in case the instruction isn’t what we think it might be. Thus, in this step we can read the two registers indicated by the rs and rt instruction fields, since it isn’t harmful to read them even if it isn’t necessary. The values read from the register file may be needed in later stages, so we read them from the register file and store the values into the temporary registers A and B.

We will also compute the branch target address with the ALU, which also is not harmful because we can ignore the value if the instruction turns out not to be a branch. The potential branch target is saved in ALUOut.

Performing these “optimistic” actions early has the benefit of decreasing the number of clock cycles needed to execute an instruction. We can do these optimistic actions early because of the regularity of the instruction formats. For instance, if the instruction has two register inputs, they are always in the rs and rt fields, and if the instruction is a branch, the offset is always the low-order 16 bits:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15:0]) << 2);
```

Operation: Access the register file to read registers rs and rt and store the results into the registers A and B. Since A and B are overwritten on every cycle, the register file can be read on every cycle with the values stored into A and B. This step also computes the branch target address and stores the address in ALUOut, where

it will be used on the next clock cycle if the instruction is a branch. This requires setting ALUSrcA to 0 (so that the PC is sent to the ALU), ALUSrcB to the value 11 (so that the sign-extended and shifted offset field is sent to the ALU), and ALUOp to 00 (so the ALU adds). The register file accesses and computation of branch target occur in parallel.

After this clock cycle, determining the action to take can depend on the instruction contents.

3. Execution, memory address computation, or branch completion

This is the first cycle during which the datapath operation is determined by the instruction class. In all cases, the ALU is operating on the operands prepared in the previous step, performing one of four functions, depending on the instruction class. We specify the action to be taken depending on the instruction class:

Memory reference:

```
ALUOut <= A + sign-extend (IR[15:0]);
```

Operation: The ALU is adding the operands to form the memory address. This requires setting ALUSrcA to 1 (so that the first ALU input is register A) and setting ALUSrcB to 10 (so that the output of the sign extension unit is used for the second ALU input). The ALUOp signals will need to be set to 00 (causing the ALU to add).

Arithmetic-logical instruction (R-type):

```
ALUOut <= A op B;
```

Operation: The ALU is performing the operation specified by the function code on the two values read from the register file in the previous cycle. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00, which together cause the registers A and B to be used as the ALU inputs. The ALUOp signals will need to be set to 10 (so that the funct field is used to determine the ALU control signal settings).

Branch:

```
if (A == B) PC <= ALUOut;
```

Operation: The ALU is used to do the equal comparison between the two registers read in the previous step. The Zero signal out of the ALU is used to determine whether or not to branch. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00 (so that the register file outputs are the ALU inputs). The ALUOp signals will need to be set to 01 (causing the ALU to subtract) for equality testing. The PCWriteCond signal will need to be asserted to update the PC if the Zero output of the ALU is asserted. By set-

ting PCSource to 01, the value written into the PC will come from ALUOut, which holds the branch target address computed in the previous cycle. For conditional branches that are taken, we actually write the PC twice: once from the output of the ALU (during the Instruction decode/register fetch) and once from ALUOut (during the Branch completion step). The value written into the PC last is the one used for the next instruction fetch.

Jump:

```
# {x, y} is the Verilog notation for concatenation of  
bit fields x and y  
PC <= {PC [31:28], (IR[25:0]], 2'b00)};
```

Operation: The PC is replaced by the jump address. PCSource is set to direct the jump address to the PC, and PCWrite is asserted to write the jump address into the PC.

4. Memory access or R-type instruction completion step

During this step, a load or store instruction accesses memory and an arithmetic-logical instruction writes its result. When a value is retrieved from memory, it is stored into the memory data register (MDR), where it must be used on the next clock cycle.

Memory reference:

```
MDR <= Memory [ALUOut];
```

or

```
Memory [ALUOut] <= B;
```

Operation: If the instruction is a load, a data word is retrieved from memory and is written into the MDR. If the instruction is a store, then the data is written into memory. In either case, the address used is the one computed during the previous step and stored in ALUOut. For a store, the source operand is saved in B. (B is actually read twice, once in step 2 and once in step 3. Luckily, the same value is read both times, since the register number—which is stored in IR and used to read from the register file—does not change.) The signal MemRead (for a load) or MemWrite (for store) will need to be asserted. In addition, for loads and stores, the signal IorD is set to 1 to force the memory address to come from the ALU, rather than the PC. Since MDR is written on every clock cycle, no explicit control signal need be asserted.

Arithmetic-logical instruction (R-type):

$\text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut};$

Operation: Place the contents of ALUOut, which corresponds to the output of the ALU operation in the previous cycle, into the Result register. The signal RegDst must be set to 1 to force the rd field (bits 15:11) to be used to select the register file entry to write. RegWrite must be asserted, and MemtoReg must be set to 0 so that the output of the ALU is written, as opposed to the memory data output.

5. Memory read completion step

During this step, loads complete by writing back the value from memory.

Load:

$\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR};$

Operation: Write the load data, which was stored into MDR in the previous cycle, into the register file. To do this, we set MemtoReg = 1 (to write the result from memory), assert RegWrite (to cause a write), and we make RegDst = 0 to choose the rt (bits 20:16) field as the register number.

This five-step sequence is summarized in Figure 5.30. From this sequence we can determine what the control must do on each clock cycle.

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$\text{IR} \leftarrow \text{Memory}[\text{PC}]$ $\text{PC} \leftarrow \text{PC} + 4$			
Instruction decode/register fetch	$A \leftarrow \text{Reg}[\text{IR}[25:21]]$ $B \leftarrow \text{Reg}[\text{IR}[20:16]]$ $\text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$\text{ALUOut} \leftarrow A \text{ op } B$	$\text{ALUOut} \leftarrow A + \text{sign-extend}(\text{IR}[15:0])$	if (A == B) $\text{PC} \leftarrow \text{ALUOut}$	$\text{PC} \leftarrow \{\text{PC}[31:28], (\text{IR}[25:0], 2'b00)\}$
Memory access or R-type completion	$\text{Reg}[\text{IR}[15:11]] \leftarrow \text{ALUOut}$	Load: $\text{MDR} \leftarrow \text{Memory}[\text{ALUOut}]$ or Store: $\text{Memory}[\text{ALUOut}] \leftarrow B$		
Memory read completion		Load: $\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR}$		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

microprogram A symbolic representation of control in the form of instructions, called microinstructions, that are executed on a simple micromachine.

Defining the Control

Now that we have determined what the control signals are and when they must be asserted, we can implement the control unit. To design the control unit for the single-cycle datapath, we used a set of truth tables that specified the setting of the control signals based on the instruction class. For the multicycle datapath, the control is more complex because the instruction is executed in a series of steps. The control for the multicycle datapath must specify both the signals to be set in any step and the next step in the sequence.

In this subsection and in [Section 5.7](#), we will look at two different techniques to specify the control. The first technique is based on finite state machines that are usually represented graphically. The second technique, called **microprogramming**, uses a programming representation for control. Both of these techniques represent the control in a form that allows the detailed implementation—using gates, ROMs, or PLAs—to be synthesized by a CAD system. In this chapter, we will focus on the design of the control and its representation in these two forms.

[Section 5.8](#) shows how hardware design languages are used to design modern processors with examples of both the multicycle datapath and the finite state control. In modern digital systems design, the final step of taking a hardware description to actual gates is handled by logic and datapath synthesis tools. Appendix C shows how this process operates by translating the multicycle control unit to a detailed hardware implementation. The key ideas of control can be grasped from this chapter without examining the material in either [Section 5.8](#) or [Appendix C](#). However, if you want to actually do some hardware design, Section 5.9 is useful, and [Appendix C](#) can show you what the implementations are likely to look like at the gate level.

Given this implementation, and the knowledge that each state requires 1 clock cycle, we can find the CPI for a typical instruction mix.

EXAMPLE

ANSWER

CPI in a Multicycle CPU

Using the SPECINT2000 instruction mix shown in Figure 3.26, what is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

The mix is 25% loads (1% load byte + 24% load word), 10% stores (1% store byte + 9% store word), 11% branches (6% beq, 5% bne), 2% jumps (1% jal + 1% jr), and 52% ALU (all the rest of the mix, which we assume to be ALU instructions). From Figure 5.30 on page 329, the number of clock cycles for each instruction class is the following:

- Loads: 5
- Stores: 4
- ALU instructions: 4
- Branches: 3
- Jumps: 3

The CPI is given by the following:

$$\begin{aligned} \text{CPI} &= \frac{\text{CPU clock cycles}}{\text{Instruction count}} = \frac{\sum \text{Instruction count}_i \times \text{CPI}_i}{\text{Instruction count}} \\ &= \sum \frac{\text{Instruction count}_i}{\text{Instruction count}} \times \text{CPI}_i \end{aligned}$$

The ratio

$$\frac{\text{Instruction count}_i}{\text{Instruction count}}$$

is simply the instruction frequency for the instruction class i . We can therefore substitute to obtain

$$\text{CPI} = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

This CPI is better than the worst-case CPI of 5.0 when all the instructions take the same number of clock cycles. Of course, overheads in both designs may reduce or increase this difference. The multicycle design is probably also more cost-effective, since it uses fewer separate components in the datapath.

The first method we use to specify the multicycle control is a **finite state machine**. A finite state machine consists of a set of states and directions on how to change states. The directions are defined by a **next-state function**, which maps the current state and the inputs to a new state. When we use a finite state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite state machine usually assumes that all outputs that are not explicitly asserted are deasserted. Similarly, the correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care. For example, the RegWrite signal should be asserted only when a register file entry is to be written; when it is not explicitly asserted, it must be deasserted.

finite state machine A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function A combinational function that, given the inputs and the current state, determines the next state of a finite state machine.

Multiplexor controls are slightly different, since they select one of the inputs whether they are 0 or 1. Thus, in the finite state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite state machine with logic, setting a control to 0 may be the default and thus may not require any gates. A simple example of a finite state machine appears in Appendix B, and if you are unfamiliar with the concept of a finite state machine, you may want to examine [Appendix B](#) before proceeding.

The finite state control essentially corresponds to the five steps of execution shown on pages 325 through 329; each state in the finite state machine will take 1 clock cycle. The finite state machine will consist of several parts. Since the first two steps of execution are identical for every instruction, the initial two states of the finite state machine will be common for all instructions. Steps 3 through 5 differ, depending on the opcode. After the execution of the last step for a particular instruction class, the finite state machine will return to the initial state to begin fetching the next instruction.

Figure 5.31 shows this abstracted representation of the finite state machine. To fill in the details of the finite state machine, we will first expand the instruction fetch and decode portion, and then we will show the states (and actions) for the different instruction classes.

We show the first two states of the finite state machine in Figure 5.32 using a traditional graphic representation. We number the states to simplify the explanation, though the numbers are arbitrary. State 0, corresponding to step 1, is the starting state of the machine.

The signals that are asserted in each state are shown within the circle representing the state. The arcs between states define the next state and are labeled with

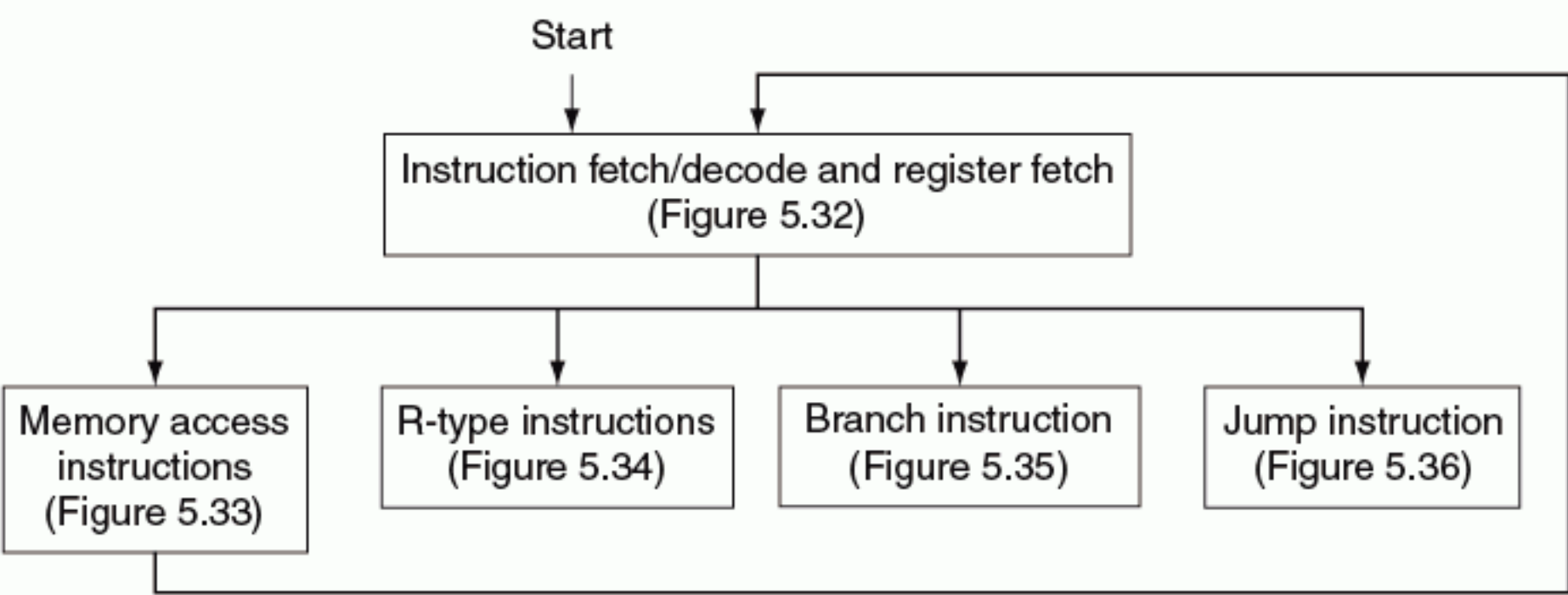


FIGURE 5.31 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

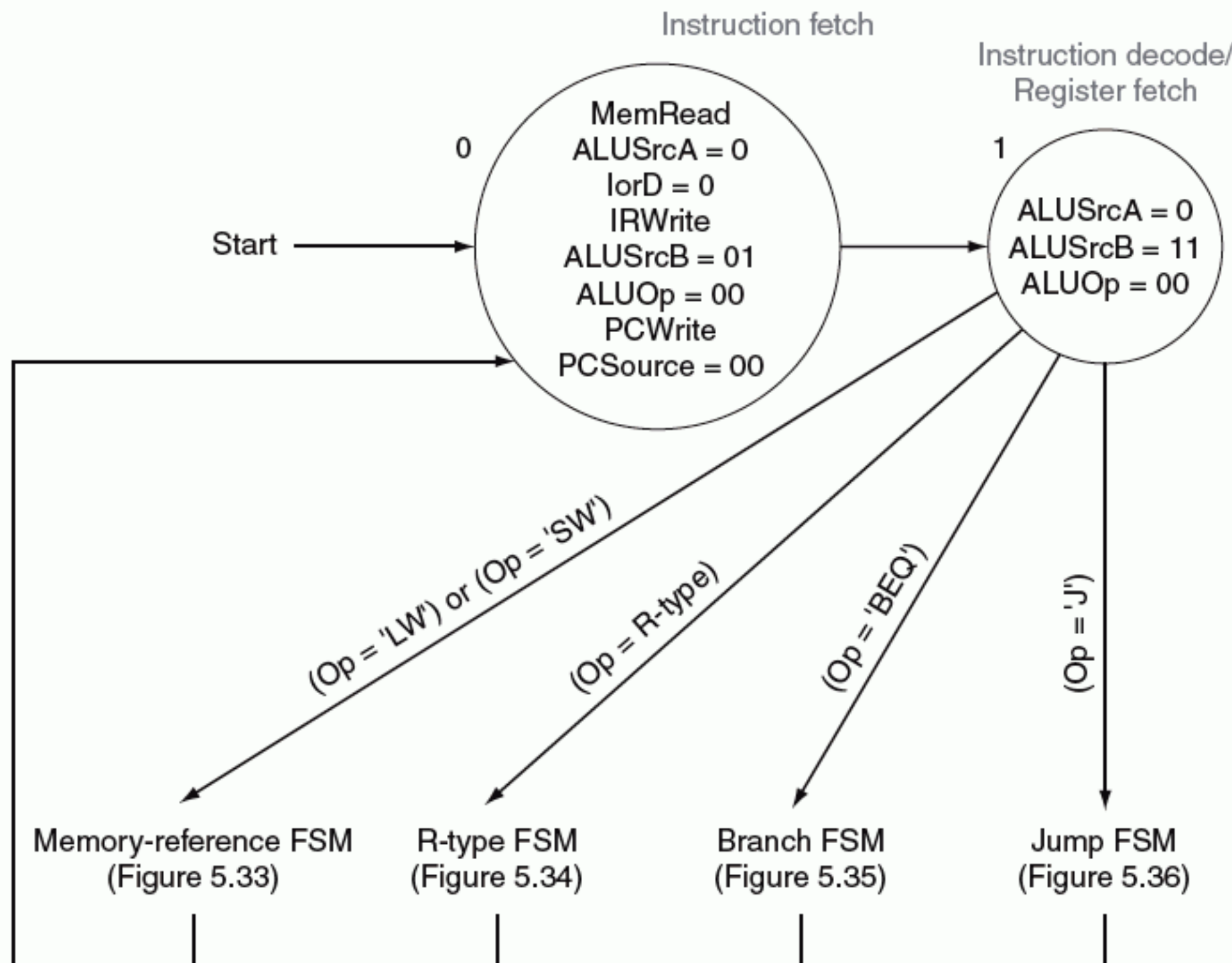


FIGURE 5.32 The instruction fetch and decode portion of every instruction is identical. These states correspond to the top box in the abstract finite state machine in Figure 5.31. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute $PC + 4$ and store it into the PC. (It will also be stored into ALUOut, but never used from there.) In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow. Remember that all signals not explicitly asserted are deasserted; this is particularly important for signals that control writes. For multiplexors controls, lack of a specific setting indicates that we do not care about the setting of the multiplexor.

conditions that select a specific next state when multiple next states are possible. After state 1, the signals asserted depend on the class of instruction. Thus, the finite state machine has four arcs exiting state 1, corresponding to the four instruction classes: memory reference, R-type, branch on equal, and jump. This process of branching to different states depending on the instruction is called *decoding*, since the choice of the next state, and hence the actions that follow, depend on the instruction class.

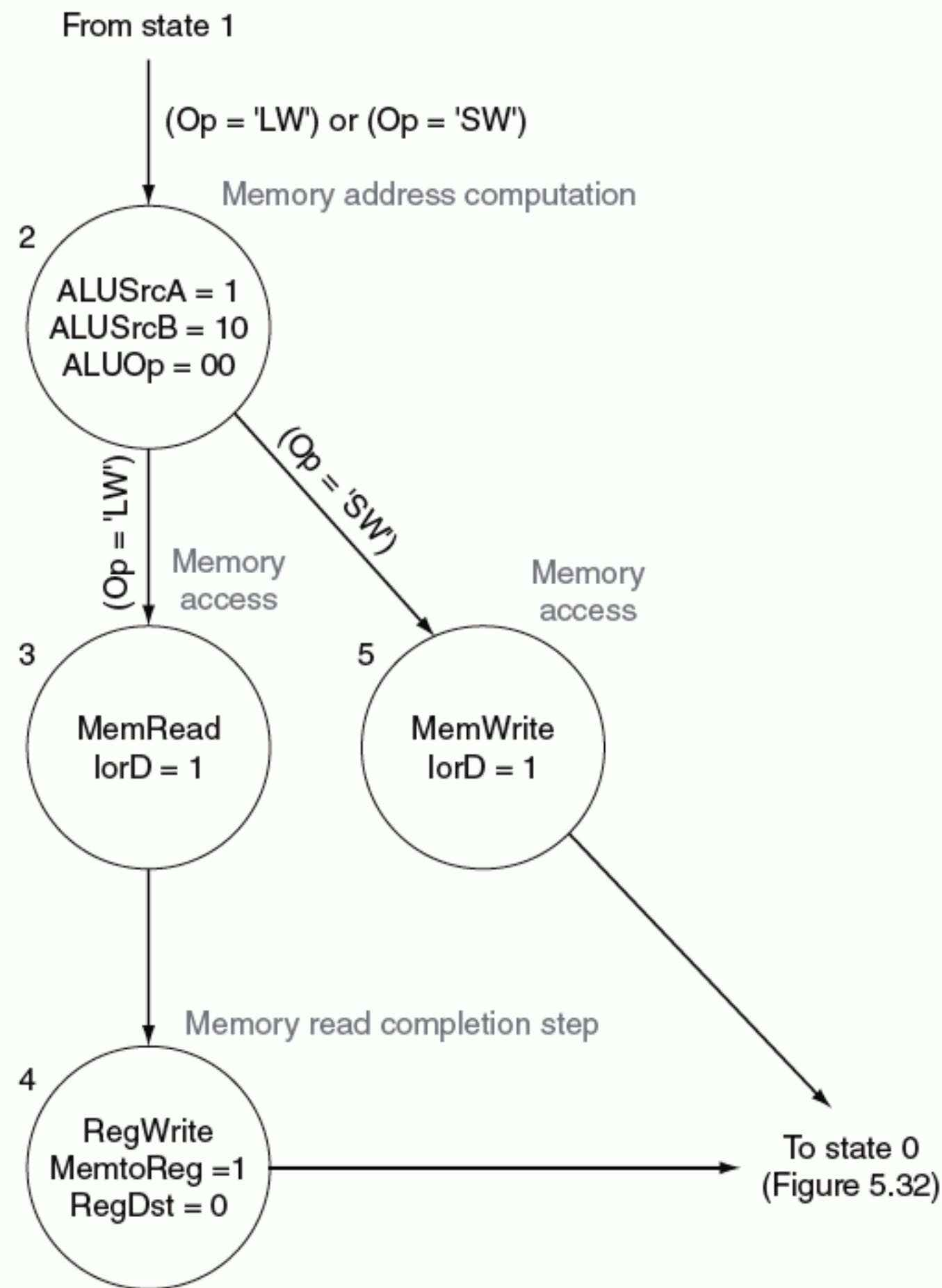


FIGURE 5.33 The finite state machine for controlling memory-reference instructions has four states. These states correspond to the box labeled “Memory access instructions” in Figure 5.31. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

Figure 5.33 shows the portion of the finite state machine needed to implement the memory-reference instructions. For the memory-reference instructions, the first state after fetching the instruction and registers computes the memory address (state 2). To compute the memory address, the ALU input multiplexors must be set so that the first input is the A register, while the second input is the sign-extended displacement field; the result is written into the ALUOut register. After the memory address calculation, the memory should be read or written; this requires two different states. If the instruction opcode is lw, then state 3 (corre-

sponding to the step Memory access) does the memory read (MemRead is asserted). The output of the memory is always written into MDR. If it is sw, state 5 does a memory write (MemWrite is asserted). In states 3 and 5, the signal IorD is set to 1 to force the memory address to come from the ALU. After performing a write, the instruction sw has completed execution, and the next state is state 0. If the instruction is a load, however, another state (state 4) is needed to write the result from the memory into the register file. Setting the multiplexor controls MemtoReg = 1 and RegDst = 0 will send the loaded value in the MDR to be written into the register file, using rt as the register number. After this state, corresponding to the Memory read completion step, the next state is state 0.

To implement the R-type instructions requires two states corresponding to steps 3 (Execute) and 4 (R-type completion). Figure 5.34 shows this two-state portion of the finite state machine. State 6 asserts ALUSrcA and sets the ALUSrcB

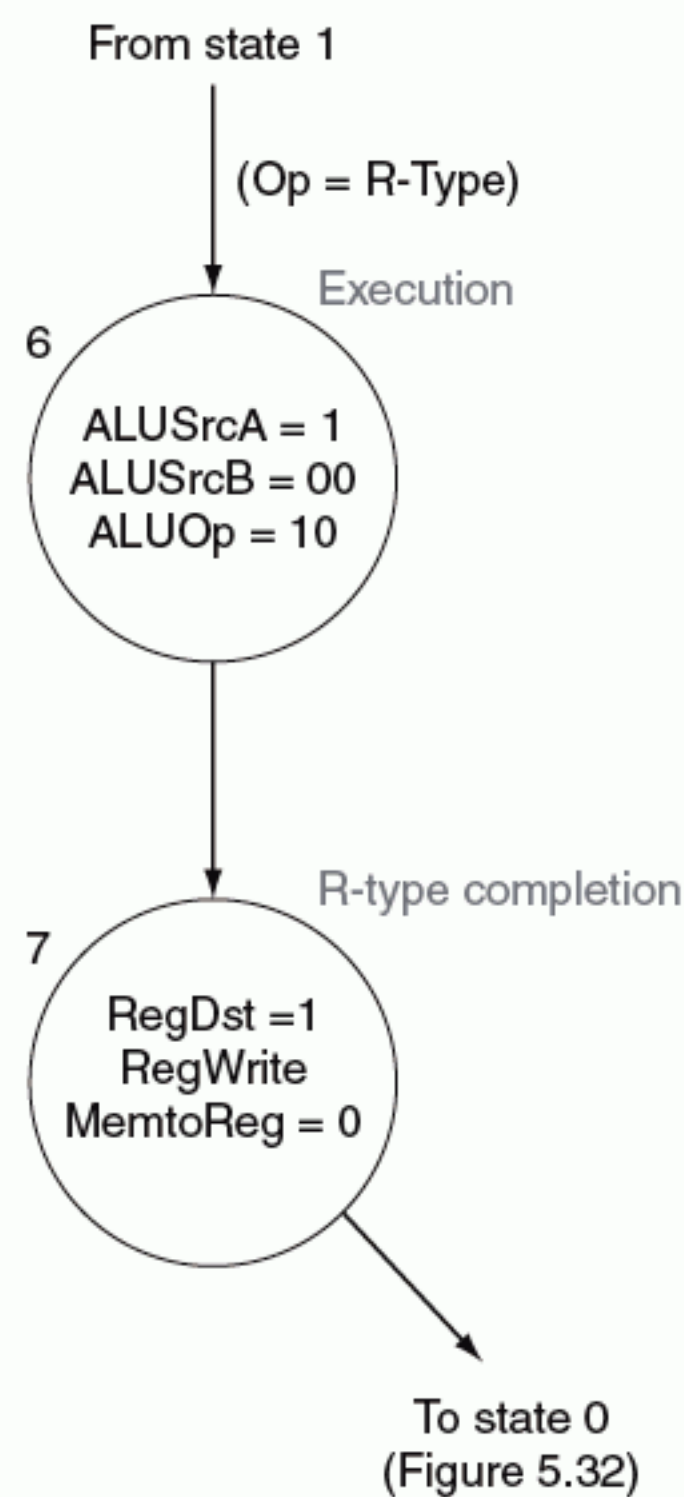


FIGURE 5.34 R-type instructions can be implemented with a simple two-state finite state machine. These states correspond to the box labeled “R-type instructions” in Figure 5.31. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

signals to 00; this forces the two registers that were read from the register file to be used as inputs to the ALU. Setting ALUOp to 10 causes the ALU control unit to use the function field to set the ALU control signals. In state 7, RegWrite is asserted to cause the register file to write, RegDst is asserted to cause the rd field to be used as the register number of the destination, and MemtoReg is deasserted to select ALUOut as the source of the value to write into the register file.

For branches, only a single additional state is necessary because they complete execution during the third step of instruction execution. During this state, the control signals that cause the ALU to compare the contents of registers A and B must be set, and the signals that cause the PC to be written conditionally with the address in the ALUOut register are also set. To perform the comparison requires that we assert ALUSrcA and set ALUSrcB to 00, and set the ALUOp value to 01 (forcing a subtract). (We use only the Zero output of the ALU, not the result of the subtraction.) To control the writing of the PC, we assert PCWriteCond and set PCSource = 01, which will cause the value in the ALUOut register (containing the branch address calculated in state 1, Figure 5.32 on page 333) to be written into the PC if the Zero bit out of the ALU is asserted. Figure 5.35 shows this single state.

The last instruction class is jump; like branch, it requires only a single state (shown in Figure 5.36) to complete its execution. In this state, the signal PCWrite is asserted to cause the PC to be written. By setting PCSource to 10, the value supplied for writing will be the lower 26 bits of the Instruction register with 00_{two} added as the low-order bits concatenated with the upper 4 bits of the PC.

We can now put these pieces of the finite state machine together to form a specification for the control unit, as shown in Figure 5.38. In each state, the signals that are asserted are shown. The next state depends on the opcode bits of the instruction, so we label the arcs with a comparison for the corresponding instruction opcodes.

A finite state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the datapath signals to be asserted as well as the next state. Figure 5.37 shows how such an implementation might look. [Appendix C](#) describes in detail how the finite state machine is implemented using this structure. In [Section C.3](#), the combinational control logic for the finite state machine of Figure 5.38 is implemented both with a ROM (read-only memory) and a PLA (programmable logic array). (Also see [Appendix B](#) for a description of these logic elements.) In the next section of this chapter, we consider another way to represent control. Both of these techniques are simply different representations of the same control information.

Pipelining, which is the subject of Chapter 6, is almost always used to accelerate the execution of instructions. For simple instructions, pipelining is capable of achieving the higher clock rate of a multicycle design and a single-cycle CPI of a single-clock design. In most pipelined processors, however, some instructions take longer than a single cycle and require multicycle control. Floating point-

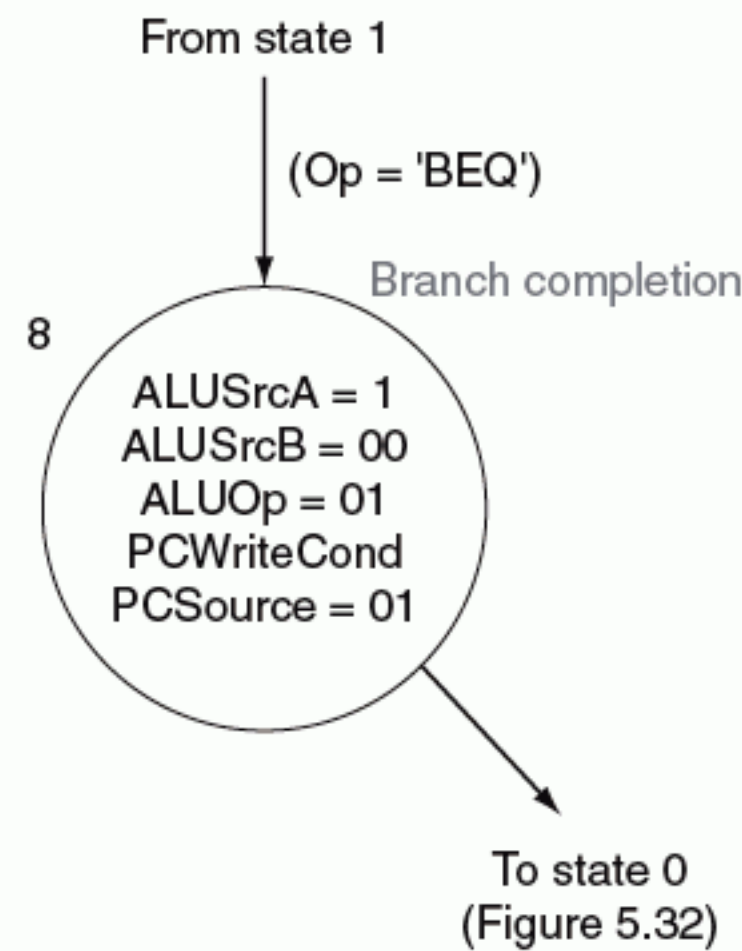


FIGURE 5.35 The branch instruction requires a single state. The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

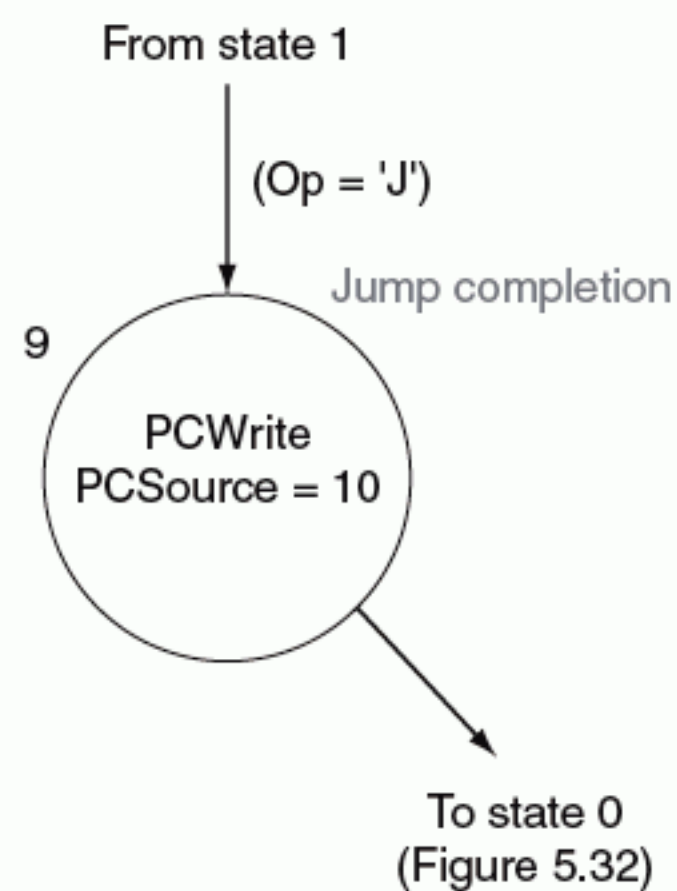


FIGURE 5.36 The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the Instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

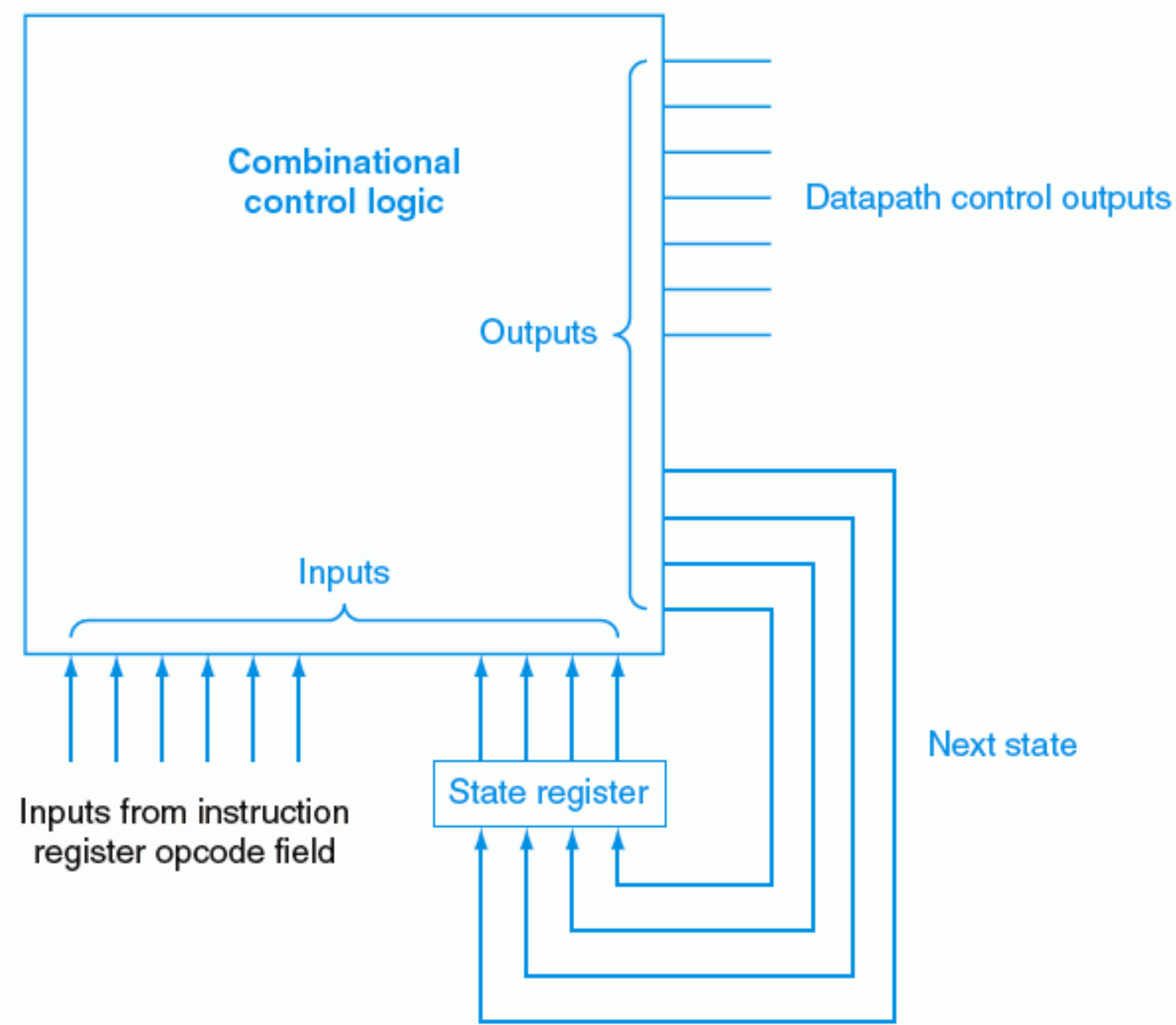


FIGURE 5.37 Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits. Notice that in the finite state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The Elaboration above explains this in more detail.

instructions are one universal example. There are many examples in the IA-32 architecture that require the use of multicycle control.

Elaboration: The style of finite state machine in Figure 5.37 is called a Moore machine, after Edward Moore. Its identifying characteristic is that the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has only the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In [Appendix C](#), when the implementation of this finite state machine is taken down to logic gates, the size advantage can be clearly seen. The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations

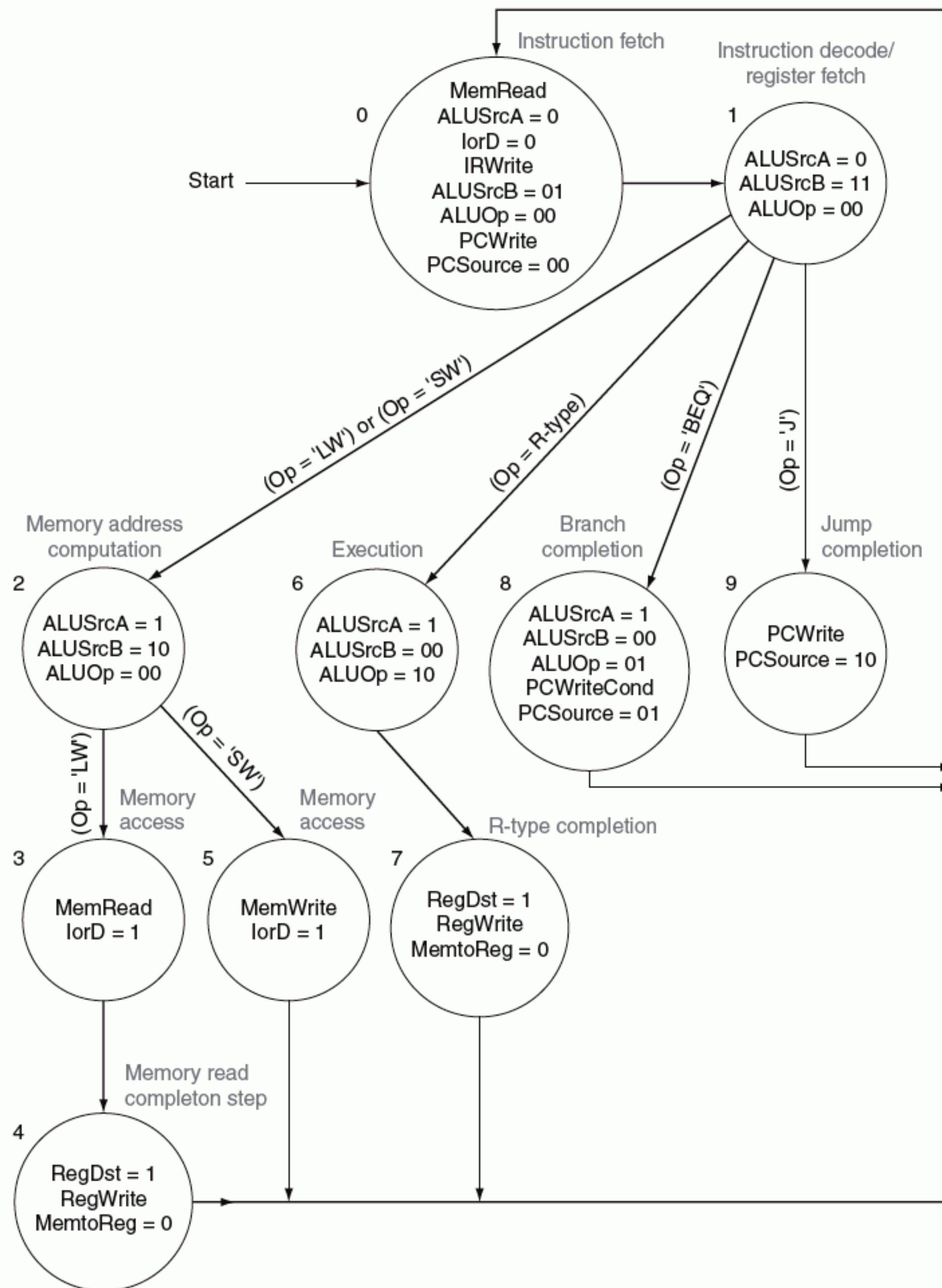


FIGURE 5.38 The complete finite state machine control for the datapath shown in Figure 5.28. The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexor control signal if the correct operation requires it. Hence, in some states a multiplexor control will be set to 0.

where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

Understanding Program Performance

For a processor with a given clock rate, the relative performance between two code segments will be determined by the product of the CPI and the instruction count to execute each segment. As we have seen here, instructions can vary in their CPI, even for a simple processor. In the next two chapters, we will see that the introduction of pipelining and the use of caches create even larger opportunities for variation in the CPI. Although many factors that affect the CPI are controlled by the hardware designer, the programmer, the compiler, and software system dictate what instructions are executed, and it is this process that determines what the effective CPI for the program will be. Programmers seeking to improve performance must understand the role of CPI and the factors that affect it.

Check Yourself

1. True or false: Since the jump instruction does not depend on the register values or on computing the branch target address, it can be completed during the second state, rather than waiting until the third.
2. True, false, or maybe: The control signal PCWriteCond can be replaced by PCSource[0].

5.6 Exceptions

exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of control is implementing **exceptions** and **interrupts**—events other than branches or jumps that change the normal flow of instruction execution. An exception is an unexpected event from within the processor; arithmetic overflow is an example of an exception. An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor. Interrupts are used by I/O devices to communicate with the processor, as we will see in Chapter 8.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal

or external; we use the term *interrupt* only when the event is externally caused. The Intel IA-32 architecture uses the word *interrupt* for all these events.

Interrupts were initially created to handle unexpected events like arithmetic overflow and to signal requests for service from I/O devices. The same basic mechanism was extended to handle internally generated exceptions as well. Here are some examples showing whether the situation is generated internally by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 7, when we discuss memory hierarchies, and in Chapter 8, when we discuss I/O, and we better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a machine, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. The basic action that the machine must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of the exception.

of the program. In Chapter 7, we will look more closely at the issue of restarting the execution.

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception. The method used in the MIPS architecture is to include a status register (called the *Cause register*), which holds a field that indicates the reason for the exception.

A second method is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	C000 0000 _{hex}
Arithmetic overflow	C000 0020 _{hex}

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending the finite state machine. Let’s assume that we are implementing the exception system used in the MIPS architecture. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to the datapath:

- *EPC*: A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- *Cause*: A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume that the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction = 0 and arithmetic overflow = 1.

We will need to add two control signals to cause the EPC and Cause registers to be written; call these *EPCWrite* and *CauseWrite*. In addition, we will need a 1-bit control signal to set the low-order bit of the Cause register appropriately; call this signal *IntCause*. Finally, we will need to be able to write the *exception address*, which is the operating system entry point for exception handling, into the PC; in

the MIPS architecture, this address is $8000\ 0180_{\text{hex}}$. (The SPIM simulator for MIPS uses $8000\ 0080_{\text{hex}}$.) Currently, the PC is fed from the output of a three-way multiplexor, which is controlled by the signal PCSource (see Figure 5.28 on page 323). We can change this to a four-way multiplexor, with additional input wired to the constant value $8000\ 0180_{\text{hex}}$. Then PCSource can be set to 11_{two} to select this value to be written into the PC.

Because the PC is incremented during the first cycle of every instruction, we cannot just write the value of the PC into the EPC, since the value in the PC will be the instruction address plus 4. However, we can use the ALU to subtract 4 from the PC and write the output into the EPC. This requires no additional control signals or paths, since we can use the ALU to subtract, and the constant 4 is already a selectable ALU input. The data write port of the EPC, therefore, is connected to the ALU output. Figure 5.39 shows the multicycle datapath with these additions needed for implementing exceptions.

Using the datapath of Figure 5.39, the action to be taken for each different type of exception can be handled in one state apiece. In each case, the state sets the Cause register, computes and saves the original PC into the EPC, and writes the exception address into the PC. Thus, to handle the two exception types we are considering, we will need to add only the two states, but before we add them we must determine how to check for exceptions, since these checks will control the arcs to the new states.

How Control Checks for Exceptions

Now we have to design a method to detect these exceptions and to transfer control to the appropriate state in the exception states. Figure 5.40 shows the two new states (10 and 11) as well as their connection to the rest of the finite state control. Each of the two possible exceptions is detected differently:

- *Undefined instruction:* This is detected when no next state is defined from state 1 for the op value. We handle this exception by defining the next-state value for all op values other than lw, sw, 0 (R-type), j, and beq as state 10. We show this by symbolically using *other* to indicate that the op field does not match any of the opcodes that label arcs out of state 1 to the new state 10, which is used for this exception.
- *Arithmetic overflow:* The ALU, designed in [Appendix B](#), included logic to detect overflow, and a signal called *Overflow* is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state (state 11) for state 7, as shown in Figure 5.40.

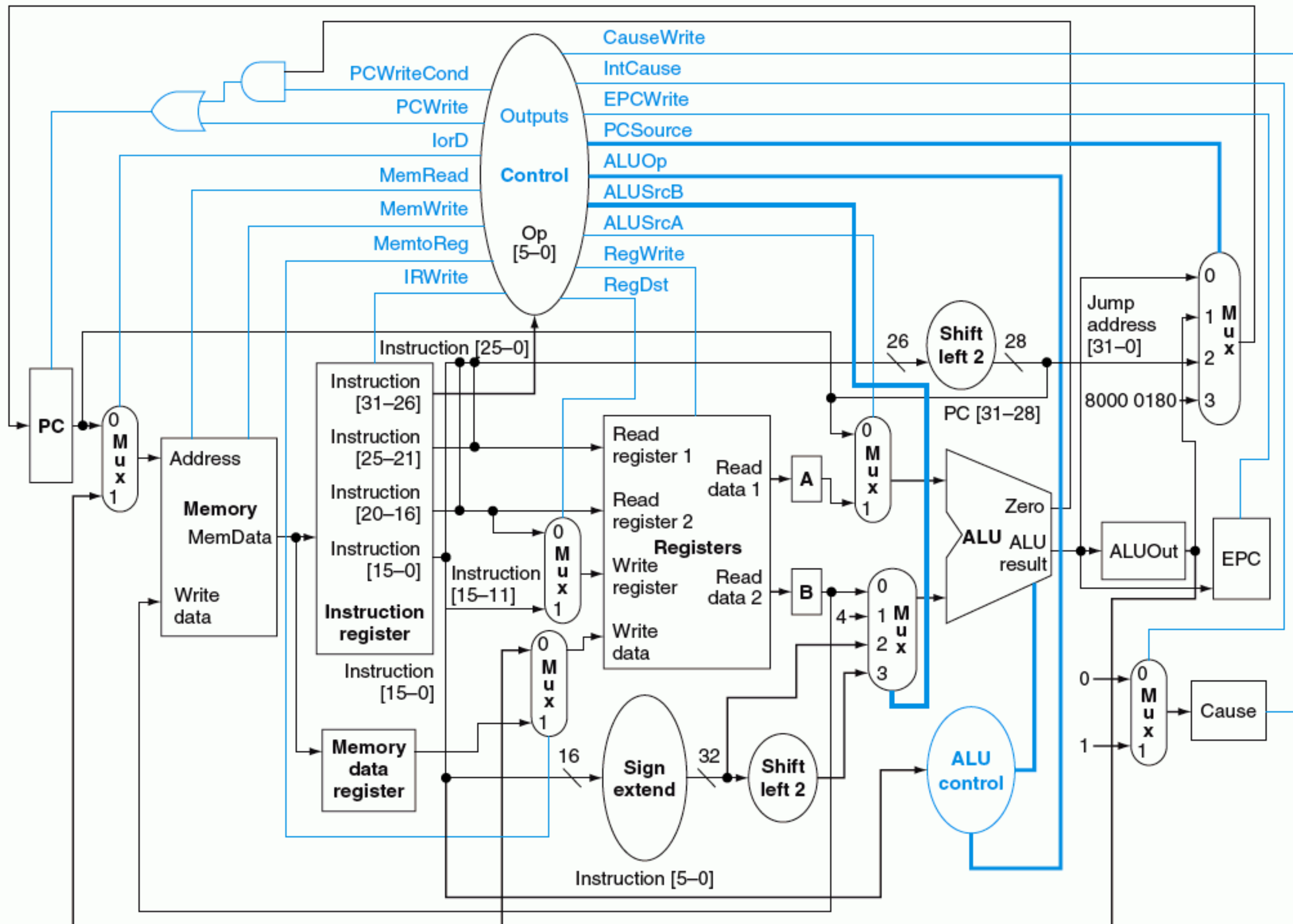


FIGURE 5.39 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.40 to see how it is used).

Figure 5.40 represents a complete specification of the control for this MIPS subset with two types of exceptions. Remember that the challenge in designing the control of a real machine is to handle the variety of different interactions between instructions and other exception-causing events in such a way that the control logic remains both small and fast. The complex interactions that are possible are what make the control unit the most challenging aspect of hardware design.

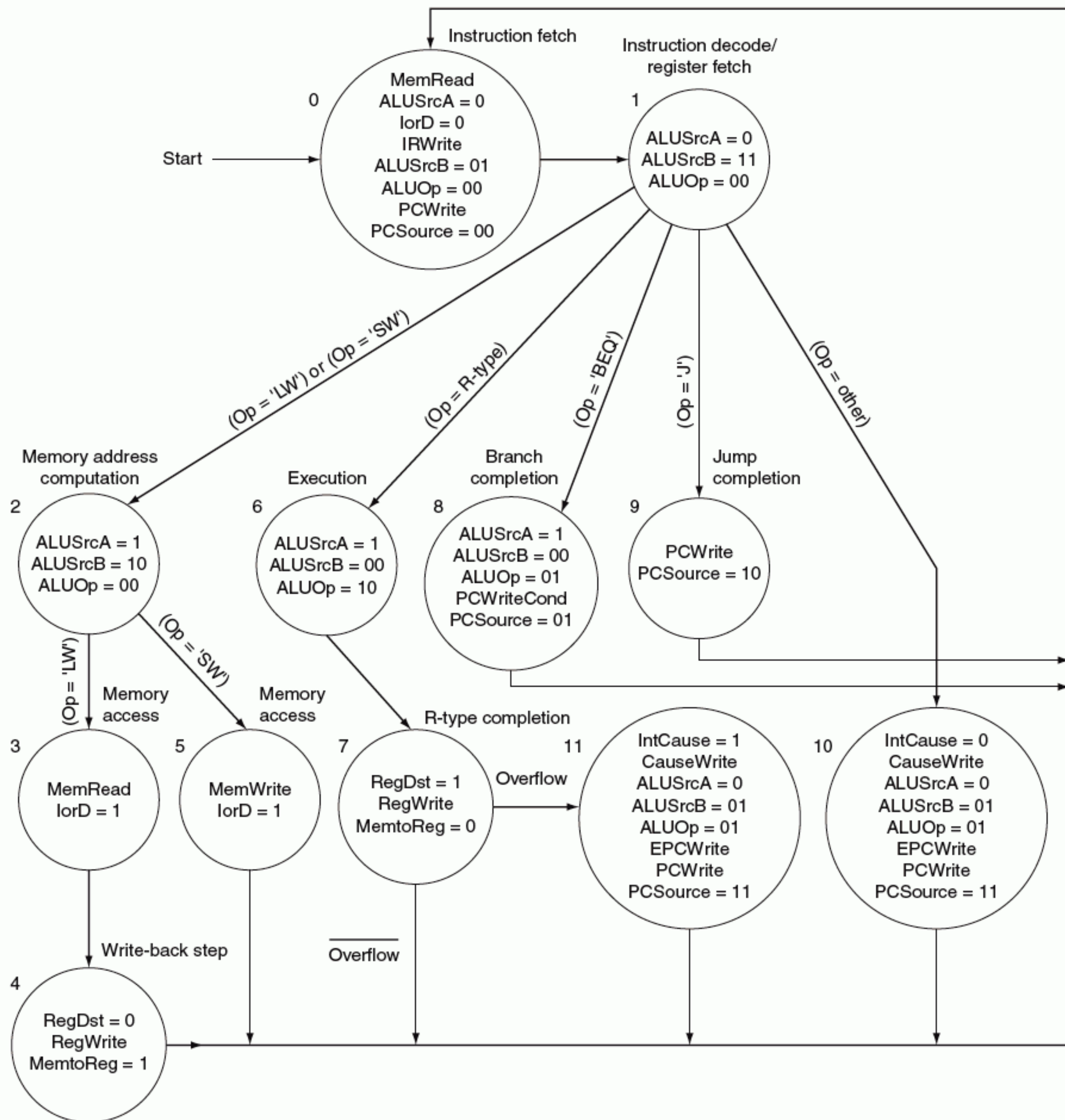


FIGURE 5.40 This shows the finite state machine with the additions to handle exception detection. States 10 and 11 are the new states that generate the appropriate control for exceptions. The branch out of state 1 labeled (*Op = other*) indicates the next state when the input does not match the opcode of any of lw, sw, 0 (R-type), j, or beq. The branch out of state 7 labeled *Overflow* indicates the action to be taken when the ALU signals an overflow.

Elaboration: If you examine the finite state machine in Figure 5.40 closely, you can see that some problems could occur in the way the exceptions are handled. For example, in the case of arithmetic overflow, the instruction causing the overflow completes writing its result because the overflow branch is in the state when the write completes. However, it's possible that the architecture defines the instruction as having no effect if the instruction causes an exception; this is what the MIPS instruction set architecture specifies. In Chapter 7, we will see that certain classes of exceptions require us to prevent the instruction from changing the machine state, and that this aspect of handling exceptions becomes complex and potentially limits performance.

Check Yourself

Is this optimization proposed in the Check Yourself on page 340 concerning PCSource still valid in the extended control for exceptions shown in Figure 5.40 on page 345? Why or why not?



Microprogramming: Simplifying Control Design

Microprogramming is a technique for designing complex control units. It uses a very simple hardware engine that can then be programmed to implement a more complex instruction set. Microprogramming is used today to implement some parts of a complex instruction set, such as a Pentium, as well as in special-purpose processors. This section, which appears on the CD, explains the basic concepts and shows how they can be used to implement the MIPS multicycle control.



An Introduction to Digital Design Using a Hardware Design Language

Modern digital design is done using hardware description languages and modern computer-aided synthesis tools that can create detailed hardware designs from the descriptions using both libraries and logic synthesis. Entire books are written on such languages and their use in digital design. This section, which appears on the CD, gives a brief introduction and shows how a hardware design language, Verilog in this case, can be used to describe the MIPS multicycle control both behaviorally and in a form suitable for hardware synthesis.

5.9

Real Stuff: The Organization of Recent Pentium Implementations

The techniques described in this chapter for building datapaths and control units are at the heart of every computer. All recent computers, however, go beyond the techniques of this chapter and use pipelining. *Pipelining*, which is the subject of the next chapter, improves performance by overlapping the execution of multiple instructions, achieving throughput close to one instruction per clock cycle (like our single-cycle implementation) with a clock cycle time determined by the delay of individual functional units rather than the entire execution path of an instruction (like our multicycle design). The last Intel IA-32 processor without pipelining was the 80386, introduced in 1985; the very first MIPS processor, the R2000, also introduced in 1985, was pipelined.

Recent Intel IA-32 processors (the Pentium II, III, and 4) employ sophisticated pipelining approaches. These processors, however, are still faced with the challenge of implementing control for the complex IA-32 instruction set, described in Chapter 2. The basic functional units and datapaths in use in modern processors, while significantly more complex than those described in this chapter, have the same basic functionality and similar types of control signals. Thus the task of designing a control unit builds on the same principles used in this chapter.

Challenges Implementing More Complex Architectures

Unlike the MIPS architecture, the IA-32 architecture contains instructions that are very complex and can take tens, if not hundreds, of cycles to execute. For example, the string move instruction (MOVS) requires calculating and updating two different memory addresses as well as loading and storing a byte of the string. The larger number and greater complexity of addressing modes in the IA-32 architecture complicates implementation of even simple instructions similar to those on MIPS. Fortunately, a multicycle datapath is well structured to adapt to variations in the amount of work required per instruction that are inherent in IA-32 instructions. This adaptability comes from two capabilities:

1. A multicycle datapath allows instructions to take varying numbers of clock cycles. Simple IA-32 instructions that are similar to those in the MIPS architecture can execute in 3 or 4 clock cycles, while more complex instructions can take tens of cycles.
2. A multicycle datapath can use the datapath components more than once per instruction. This is critical to handling more complex addressing modes, as well as implementing more complex operations, both of which are present in the IA-32 architecture. Without this capability, the datapath

microprogrammed control A method of specifying control that uses microcode rather than a finite state representation.

hardwired control An implementation of finite state machine control typically using programmable logic arrays (PLAs) or collections of PLAs and random logic.

microcode The set of microinstructions that control a processor.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

microinstruction A representation of control using low-level instructions, each of which asserts a set of control signals that are active on a given clock cycle as well as specifies what microinstruction to execute next.

micro-operations The RISC-like instructions directly executed by the hardware in recent Pentium implementations.

would need to be extended to handle the demands of the more complex instructions without reusing components, which would be completely impractical. For example, a single-cycle datapath, which doesn't reuse components, for the IA-32 would require several data memories and a very large number of ALUs.

Using the multicycle datapath and a **microprogrammed controller** provides a framework for implementing the IA-32 instruction set. The challenging task, however, is creating a high-performance implementation, which requires dealing with the diversity of the requirements arising from different instructions. Simply put, a high-performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize primarily the complex, less frequently used, instructions.

To accomplish this goal, every Intel implementation of the IA-32 architecture since the 486 has used a combination of **hardwired control** to handle simple instructions, and **microcoded** control to handle the more complex instructions. For those instructions that can be executed in a single pass through the datapath—those with complexity similar to a MIPS instruction—the hardwired control generates the control information and executes the instruction in one pass through the datapath that takes a small number of clock cycles. Those instructions that require multiple datapath passes and complex sequencing are handled by the microcoded controller that takes a larger number of cycles and multiple passes through the datapath to complete the execution of the instruction. The benefit of this approach is that it enables the designer to achieve low cycle counts for the simple instructions without having to build the enormously complex datapath that would be required to handle the full generality of the most complex instructions.

The Structure of the Pentium 4 Implementation

Recent Pentium processors are capable of executing more than one instruction per clock, using an advanced pipelining technique, called **superscalar**. We describe how a superscalar processor works in the next chapter. The important thing to understand here is that executing more than one instruction per clock requires duplicating the datapath resources. The simplest way to think about this is that the processor has multiple datapaths, although these are tailored to handle one class of instructions: say, loads and stores, ALU operations, or branches. In this way, the processor is able to execute a load or store in the same clock cycle that it is also executing a branch and an ALU operation. The Pentium III and 4 allow up to three IA-32 instructions to execute in a clock cycle.

The Pentium III and Pentium 4 execute simple **microinstructions** similar to MIPS instructions, called **micro-operations** in Intel terminology. These microinstructions are fully self-contained operations that are initially about 70 bits wide. The control of datapath to implement these microinstructions is completely hard-

wired. This last level of control expands up to three microinstructions into about 120 control lines for the integer datapaths and 275 to over 400 control lines for the floating-point datapath—the latter number for the new SSE2 instructions included in the Pentium 4. This last step of expanding the microinstructions into control lines is very similar to the control generation for the single-cycle datapath or for the ALU control.

How is the translation between IA-32 instructions and microinstructions performed? In earlier Pentium implementations (i.e., the Pentium Pro, Pentium II, and Pentium III), the instruction decode unit would look at up to three IA-32 instructions at a time and use a set of PLAs to generate up to six microinstructions per cycle. With the significantly higher clock rate introduced in the Pentium 4, this solution was no longer adequate and an entirely new method of generating microinstructions was needed.

The solution adopted in the Pentium 4 is to include a **trace cache** of microinstructions, which is accessed by the IA-32 program counter. A trace cache is a sophisticated form of instruction cache, which we explain in detail in Chapter 7. For now, think of it as a buffer that holds the microinstructions that implement a given IA-32 instruction. When the trace cache is accessed with the address of the next IA-32 instruction to be executed, one of several events occurs:

- The translation of the IA-32 instruction is in the trace cache. In this case, up to three microinstructions are produced from the trace cache. These three microinstructions represent from one to three IA-32 instructions. The IA-32 PC is advanced one to three instructions depending on how many fit in the three microinstruction sequence.
- The translation of the IA-32 instruction is in the trace cache, but it requires more than four microinstructions to implement. For such complex IA-32 instructions, there is a microcode ROM; the control unit transfers control to the microprogram residing in the ROM. Microinstructions are produced from the microprogram until the more complex IA-32 instruction has been completed. The microcode ROM provides a total of more than 8000 microinstructions, with a number of sequences being shared among IA-32 instructions. Control then transfers back to fetching instructions from the trace cache.
- The translation of the designated IA-32 instruction is not in the trace cache. In this case, an IA-32 instruction decoder is used to decode the IA-32 instruction. If the number of microinstructions is four or less, the decoded microinstructions are placed in the trace cache, where they may be found on the next execution of this instruction. Otherwise, the microcode ROM is used to complete the sequence.

From one to three microinstructions are sent from the trace cache to the Pentium 4 microinstruction pipeline, which we describe in detail at the end of Chapter 6. The use of simple low-level hardwired control and simple datapaths for

trace cache An instruction cache that holds a sequence of instructions with a given starting address; in recent Pentium implementations the trace cache holds microoperations rather than IA-32 instructions.

handling the microinstructions together with the trace cache of decoded instructions allows the Pentium 4 to achieve impressive clock rates, similar to those for microprocessors implementing simpler instruction set architectures. Furthermore, the translation process, which combines direct hardwired control for simple instructions with microcoded control for complex instructions, allows the Pentium 4 to execute the simple, high-frequency instructions in the IA-32 instruction set at a high rate, yielding a low, and very competitive, CPI.

Understanding Program Performance

dispatch An operation in a microprogrammed control unit in which the next microinstruction is selected on the basis of one or more fields of a macroinstruction, usually by creating a table containing the addresses of the target microinstructions and indexing the table using a field of the macroinstruction. The dispatch tables are typically implemented in ROM or programmable logic array (PLA). The term *dispatch* is also used in dynamically scheduled processors to refer to the process of sending an instruction to a queue.

Although most of the Pentium 4 performance, ignoring the memory system, depends on the efficiency of the pipelined microoperations, the effectiveness of the front end in decoding IA-32 instructions can have a significant effect on performance. In particular, because of the structure of the decoder, using simpler IA-32 instructions that require four or fewer microoperations, and hence, avoiding a microcode **dispatch**, is likely to lead to better performance. Because of this implementation strategy (and a similar one on the Pentium III), compiler writers and assembly language programmers should try to make use of sequences of simple IA-32 instructions rather than more complex alternatives.

5.10

Fallacies and Pitfalls

Pitfall: Adding a complex instruction implemented with microprogramming may not be faster than a sequence using simpler instructions.

Most machines with a large and complex instruction set are implemented, at least in part, using microcode stored in ROM. Surprisingly, on such machines, sequences of individual simpler instructions are sometimes as fast as or even faster than the custom microcode sequence for a particular instruction.

How can this possibly be true? At one time, microcode had the advantage of being fetched from a much faster memory than instructions in the program. Since caches came into use in 1968, microcode no longer has such a consistent edge in fetch time. Microcode does, however, still have the advantage of using internal temporary registers in the computation, which can be helpful on machines with few general-purpose registers. The disadvantage of microcode is that the algorithms must be selected before the machine is announced and can't be

changed until the next model of the architecture. The instructions in a program, on the other hand, can utilize improvements in its algorithms at any time during the life of the machine. Along the same lines, the microcode sequence is probably not optimal for all possible combinations of operands.

One example of such an instruction in the IA-32 implementations is the move string instruction (MOVS) used with a repeat prefix that we discussed in Chapter 2. This instruction is often slower than a loop that moves words at a time, as we saw earlier in the Fallacies and Pitfalls (see page 350).

Another example involves the LOOP instruction, which decrements a register and branches to the specified label if the decremented register is not equal to zero. This instruction is designed to be used as the branch at the bottom of loops that have a fixed number of iterations (e.g., many *for* loops). Such an instruction, in addition to packing in some extra work, has benefits in minimizing the potential losses from the branch in pipelined machines (as we will see when we discuss branches in the next chapter).

Unfortunately, on all recent Intel IA-32 implementations, the LOOP instruction is always slower than the macrocode sequence consisting of simpler individual instructions (assuming that the small code size difference is not a factor). Thus, optimizing compilers focusing on speed never generate the LOOP instruction. This, in turn, makes it hard to motivate making LOOP fast in future implementations, since it is so rarely used!

Fallacy: If there is space in the control store, new instructions are free of cost.

One of the benefits of a microprogrammed approach is that control store implemented in ROM is not very expensive, and as transistor budgets grew, extra ROM was practically free. The analogy here is that of building a house and discovering, near completion, that you have enough land and materials left to add a room. This room wouldn't be free, however, since there would be the costs of labor and maintenance for the life of the home. The temptation to add "free" instructions can occur only when the instruction set is not fixed, as is likely to be the case in the first model of a computer. Because upward compatibility of binary programs is a highly desirable feature, all future models of this machine will be forced to include these so-called free instructions, even if space is later at a premium.

During the design of the 80286, many instructions were added to the instruction set. The availability of more silicon resource and the use of microprogrammed implementation made such additions seem painless. Possibly the largest addition was a sophisticated protection mechanism, which is largely unused today, but still must be implemented in newer implementations. This addition was motivated by a perceived need for such a mechanism and the desire to enhance microprocessor architectures to provide functionality equal to that of

larger computers. Likewise, a number of decimal instructions were added to provide decimal arithmetic on bytes. Such instructions are rarely used today because using binary arithmetic on 32 bits and converting back and forth to decimal representation is considerably faster. Like the protection mechanisms, the decimal instructions must be implemented in newer processors even if only rarely used.

5.11 Concluding Remarks

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In Section 5.3, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense. Along the same lines, in the first portion of Section 5.5, we saw how the decision to break the clock cycle into a series of steps led to the revised multicycle datapath. In both cases, the top-level organization—a single-cycle or multicycle machine—together with the instruction set, prescribed many characteristics of the datapath design.

The BIG Picture

Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique. Figure 5.41 shows the variety of methods for specifying the control and moving from the specification to an implementation using some form of structured logic.

Similarly, the control is largely defined by the instruction set architecture, the organization, and the datapath design. In the single-cycle organization, these three aspects essentially define how the control signals must be set. In the multicycle design, the exact decomposition of the instruction execution into cycles, which is based on the instruction set architecture, together with the datapath, defines the requirements on the control.

Control is one of the most challenging aspects of computer design. A major reason is that designing the control requires an understanding of how all the com-

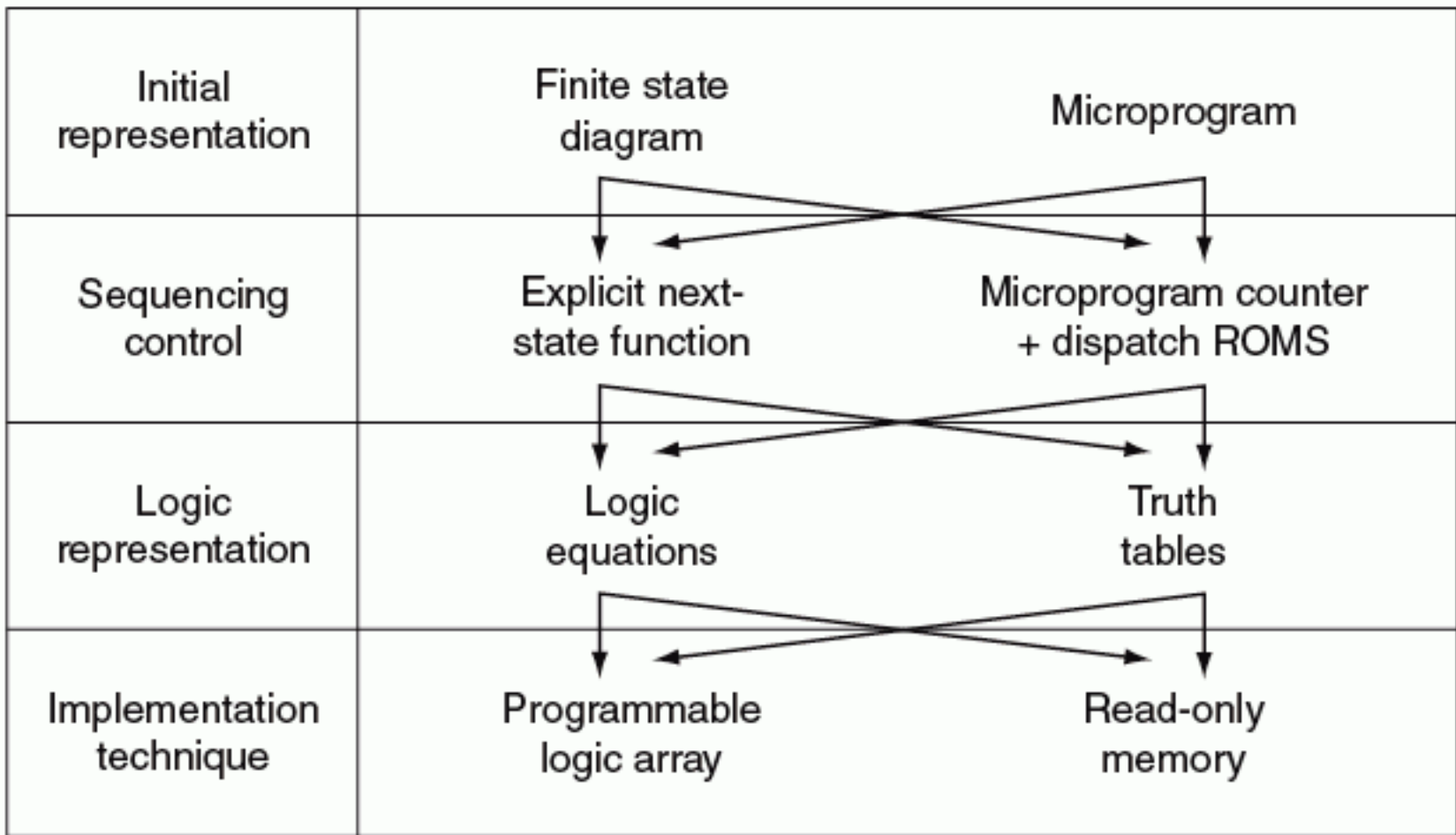


FIGURE 5.41 Alternative methods for specifying and implementing control. The arrows indicate possible design paths: any path from the initial representation to the final implementation technology is viable. Traditionally, “hardwired control” means that the techniques on the left-hand side are used, and “microprogrammed control” means that the techniques on the right-hand side are used.

ponents in the processor operate. To help meet this challenge, we examined two techniques for specifying control: finite state diagrams and microprogramming. These control representations allow us to abstract the specification of the control from the details of how to implement it. Using abstraction in this fashion is the major method we have to cope with the complexity of computer designs.

Once the control has been specified, we can map it to detailed hardware. The exact details of the control implementation will depend on both the structure of the control and on the underlying technology used to implement it. Abstracting the specification of control is also valuable because the decisions of how to implement the control are technology dependent and likely to change over time.



Historical Perspective and Further Reading

The rise of microprogramming and its effect on instruction set design and computer development is one of the more interesting interactions in the first few decades of the electronic computer. This story is the focus of the historical perspectives section on the CD.

5.13 Exercises

5.1 [6] <§5.2> Do we need combinational logic, sequential logic, or a combination of the two to implement each of the following:

- a. multiplexor
- b. comparator
- c. incrementer/decrementer
- d. barrel shifter
- e. multiplier with shifters and adders
- f. register
- g. memory
- h. ALU (the ones in single-cycle and multiple-cycle datapaths)
- i. carry look-ahead adder
- j. latch
- k. general finite state machine (FSM)

5.2 [10] <§5.4> Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have for the signals shown below, in the single-cycle datapath in Figure 5.17 on page 307. Which instructions, if any, will not work correctly? Explain why.

Consider each of the following faults separately:

- a. $\text{RegWrite} = 0$
- b. $\text{ALUOp0} = 0$
- c. $\text{ALUOp1} = 0$
- d. $\text{Branch} = 0$
- e. $\text{MemRead} = 0$
- f. $\text{MemWrite} = 0$

5.3 [5] <§5.4> This exercise is similar to Exercise 5.2, but this time consider stuck-at-1 faults (the signal is always 1).

5.4 [5] <§5.4>  **For More Practice:** Single Cycle Datapaths with Floating Point.

5.5 [5] <§5.4>  **For More Practice:** Single Cycle Datapaths with Floating Point.

5.6 [10] <§5.4>  **For More Practice:** Single Cycle Datapaths with Floating Point.

5.7 [2–3 months] <§§5.1–5.4> Using standard parts, build a machine that implements the single-cycle machine in this chapter.

5.8 [15] <§5.4> We wish to add the instruction `jr` (jump register) to the single-cycle datapath described in this chapter. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.17 on page 307 and show the necessary additions to Figure 5.18 on page 308. You can photocopy these figures to make it faster to show the additions.

5.9 [10] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `sll` (shift left logical), which is described in Section 2.5.

5.10 [15] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `lui` (load upper immediate), which is described in Section 2.9.

5.11 [20] <§5.4> This question is similar to Exercise 5.8 except that we wish to add a variant of the `lw` (load word) instruction, which increments the index register after loading word from memory. This instruction (`l_inc`) corresponds to the following two instructions:

```
lw    $rs,L($rt)
addi  $rt,$rt,1
```












5.12 [5] <§5.4> Explain why it is not possible to modify the single-cycle implementation to implement the load with increment instruction described in Exercise 5.12 without modifying the register file.

5.13 [7] <§5.4> Consider the single-cycle datapath in Figure 5.17. A friend is proposing to modify this single-cycle datapath by eliminating the control signal `MemtoReg`. The multiplexor that has `MemtoReg` as an input will instead use either the `ALUSrc` or the `MemRead` control signal. Will your friend's modification work? Can one of the two signals (`MemRead` and `ALUSrc`) substitute for the other? Explain.

5.14 [10] <§5.4> MIPS chooses to simplify the structure of its instructions. The way we implement complex instructions through the use of MIPS instructions is to decompose such complex instructions into multiple simpler MIPS ones. Show how MIPS can implement the instruction `swap $rs, $rt`, which swaps the contents of registers `$rs` and `$rt`. Consider the case in which there is an available register that may be destroyed as well as the case in which no such register exists.

If the implementation of this instruction in hardware will increase the clock period of a single-instruction implementation by 10%, what percentage of swap operations in the instruction mix would recommend implementing it in hardware?

5.15 [5] <§5.4>  **For More Practice:** Effects of Faults in Control Multiplexors

- 5.16** [5] <§5.4>  **For More Practice:** Effects of Faults in Control Multiplexors
- 5.17** [5] <§5.5>  **For More Practice:** Effects of Faults in Control Multiplexors
- 5.18** [5] <§5.5>  **For More Practice:** Effects of Faults in Control Multiplexors
- 5.19** [15] <§5.4>  **For More Practice:** Adding Instructions to the Datapath
- 5.20** [15] <§5.4>  **For More Practice:** Adding Instructions to the Datapath
- 5.21** [8] <§5.4>  **For More Practice:** Adding Instructions to the Datapath
- 5.22** [8] <§5.4>  **For More Practice:** Adding Instructions to the Datapath
- 5.23** [5] <§5.4>  **For More Practice:** Adding Instructions to the Datapath
- 5.24** [10] <§5.4>  **For More Practice:** Datapath Control Signals
- 5.25** [10] <§5.4>  **For More Practice:** Datapath Control Signals
- 5.26** [15] <§5.4>  **For More Practice:** Modifying the Datapath and Control
- 5.27** [8] <§5.4> Repeat Exercise 5.14, but apply your solution to the instruction load with increment: `l_incr $rt,Address($rs)`.
- 5.28** [5] <§5.4> The concept of the “critical path,” the longest possible path in the machine, was introduced in 5.4 on page 315. Based on your understanding of the single-cycle implementation, show which units can tolerate more delays (i.e., are not on the critical path), and which units can benefit from hardware optimization. Quantify your answers taking the same numbers presented on page 315 (Section 5.4, “Example: Performance of Single-Cycle Machines”).
- 5.29** [5] <§5.5> This exercise is similar to Exercise 5.2, but this time consider the effect that the stuck-at-0 faults would have on the *multiple-cycle* datapath in Figure 5.27. Consider each of the following faults:
- RegWrite = 0
 - MemRead = 0
 - MemWrite = 0
 - IRWrite = 0
 - PCWrite = 0
 - PCWriteCond = 0.
- 5.30** [5] <§5.5> This exercise is similar to Exercise 5.29, but this time consider stuck-at-1 faults (the signal is always 1).
- 5.31** [[15] <§§5.4, 5.5> This exercise is similar to Exercise 5.13 but more general. Determine whether any of the control signals in the single-cycle implementation

can be eliminated and replaced by another existing control signal, or its inverse. Note that such redundancy is there because we have a very small set of instructions at this point, and it will disappear (or be harder to find) when we implement a larger number of instructions.

5.32 [15] <§5.5> We wish to add the instruction `lui` (load upper immediate) described in Chapter 3 to the multicycle datapath described in this chapter. Use the same structure of the multicycle datapath of Figure 5.28 on page 323 and show the necessary modifications to the finite state machine of Figure 5.38 on page 339. You may find it helpful to examine the execution steps shown on pages 325 through 329 and consider the steps that will need to be performed to execute the new instruction. How many cycles are required to implement this instruction?

5.33 [15] <§5.5> You are asked to modify the implementation of `lui` in Exercise 5.32 in order to cut the execution time by 1 cycle. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.28 on page 323. You can photocopy existing figures to make it easier to show your modifications. You have to maintain the assumption that you don't know what the instruction is before the end of state 1 (end of second cycle). Please explicitly state how many cycles it takes to execute the new instruction on your modified datapath and finite state machine.

5.34 [20] <§5.5> This question is similar to Exercise 5.32 except that we wish to implement a new instruction `ldi` (load immediate) that loads a 32-bit immediate value from the memory location following the instruction address.

5.35 [15] <§5.5> Consider a change to the multiple-cycle implementation that alters the register file so that it has only one read port. Describe (via a diagram) any additional changes that will need to be made to the datapath in order to support this modification. Modify the finite state machine to indicate how the instructions will work, given your new datapath.

5.36 [15] <§5.5> Two important parameters control the performance of a processor: cycle time and cycles per instruction. There is an enduring trade-off between these two parameters in the design process of microprocessors. While some designers prefer to increase the processor frequency at the expense of large CPI, other designers follow a different school of thought in which reducing the CPI comes at the expense of lower processor frequency.

Consider the following machines, and compare their performance using the SPEC CPUint 2000 data from Figure 3.26 on page 228.

M1: The multicycle datapath of Chapter 5 with a 1 GHz clock.

M2: A machine like the multicycle datapath of Chapter 5, except that register updates are done in the same clock cycle as a memory read or ALU operation.

Thus in Figure 5.38 on page 339, states 6 and 7 and states 3 and 4 are combined. This machine has an 3.2 GHz clock, since the register update increases the length of the critical path.

M3: A machine like M2 except that effective address calculations are done in the same clock cycle as a memory access. Thus states 2, 3, and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 2.8 GHz clock because of the long cycle created by combining address calculation and memory access.

Find out which of the machines is fastest. Are there instruction mixes that would make another machine faster, and if so, what are they?










5.37 [20] <§5.5> Your friends at C³ (Creative Computer Corporation) have determined that the critical path that sets the clock cycle length of the multicycle datapath is memory access for loads and stores (not for fetching instructions). This has caused their newest implementation of the MIPS 30000 to run at a clock rate of 4.8 GHz rather than the target clock rate of 5.6 GHz. However, Clara at C³ has a solution. If all the cycles that access memory are broken into two clock cycles, then the machine can run at its target clock rate.

Using the SPEC CPUint 2000 mixes shown in Chapter 3 (Figure 3.26 on page 228), determine how much faster the machine with the two-cycle memory accesses is compared with the 4.8 GHz machine with single-cycle memory access. Assume that all jumps and branches take the same number of cycles and that the set instructions and arithmetic immediate instructions are implemented as R-type instructions. Would you consider the further step of splitting instruction fetch into two cycles if it would raise the clock rate up to 6.4 GHz? Why?

5.38 [20] <§5.5> Suppose there were a MIPS instruction, called `bcmp`, that compares two blocks of words in two memory addresses. Assume that this instruction requires that the starting address of the first block is in register `$t1` and the starting address of the second block is in `$t2`, and that the number of words to compare is in `$t3` (which is `$t3 ≥ 0`). Assume the instruction can leave the result (the address of the first mismatch or zero if a complete match) in `$t1` and/or `$t2`. Furthermore, assume that the values of these registers as well as registers `$t4` and `t5` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Write the MIPS assembly language program to implement (emulate the behavior of) block compare. How many instructions will be executed to compare two 100-word blocks? Using the CPI of the instructions in the multicycle implementation, how many cycles are needed for the 100-word block compare?

5.39 [2–3 months] <§§5.1–5.5> Using standard parts, build a machine that implements the multicycle machine in this chapter.

- 5.40** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.41** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.42** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.43** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.44** [15] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.45** [20] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.46** [10] <§5.5>  **For More Practice:** Adding Instructions to the Datapath
- 5.47** [15] <§§5.1–5.5>  **For More Practice:** Comparing Processor Performance
- 5.48** [20] <§5.5>  **For More Practice:** Implementing Instructions in MIPS
- 5.49** [30] <§5.6> We wish to add the instruction `eret` (exception return) to the multicycle datapath described in this chapter. A primary task of the `eret` instruction is to reload the PC with the return address at which an exception, or error trap occurred. Suppose that if the processor is serving an error trap, then the PC has to be loaded from a register `ErrorPC`. Otherwise the processor is serving an exception) the PC has to be loaded from `EPC`. Suppose that there is a bit in the cause register called `trap` to encode an error trap when it occurs and to save the PC in the `ErrorPC` register. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.39 on page 344 to accommodate the trap/exception call and return, and show the necessary modifications to the finite state machine of Figure 5.40 on page 345 to implement the `eret` instruction. You can photocopy the figures to make it easier to show your modifications.
- 5.50** [6] <§5.6> Exceptions occur when a control flow change is required to handle an unexpected event in the processor. How can the cause and the instruction that caused the exception, be represented by the hardware in a MIPS machine? Give two examples for conditions that a processor can handle by restarting execution of instructions after handling the exception, and two others for exceptions that lead to program termination.
- 5.51** [6] <§5.6> Exception detection is an important aspect of exception handling. Try to identify the cycle in which the following exceptions can be detected for the multicycle datapath in Figure 5.28 on page 323.

Consider the following exceptions:

- a. Divide by zero exception (suppose we use the same ALU for division in one cycle, and that it is recognized by the rest of the control)
- b. Overflow exception

- c. Invalid instruction
- d. External interrupt
- e. Invalid instruction memory address
- f. Invalid data memory address


5.52 [15] <§5.6>  **For More Practice:** Adding Instructions to the Datapath

5.53 [30] <§5.7> Microcode has been used to add more powerful instructions to an instruction set; let's explore the potential benefits of this approach. Devise a strategy for implementing the `bcmp` instruction described in Exercise 5.38 using the multicycle datapath and microcode. You will probably need to make some changes to the datapath in order to efficiently implement the `bcmp` instruction. Provide a description of your proposed changes and describe how the `bcmp` instruction will work. Are there any advantages that can be obtained by adding internal registers to the datapath to help support the `bcmp` instruction? Estimate the improvement in performance that you can achieve by implementing the instruction in hardware (as opposed to the software solution you obtained in Exercise 5.38) and explain where the performance increase comes from.

5.54 [30] <§5.7>  **For More Practice:** Microcode

5.55 [30] <§5.7>  **For More Practice:** Microcode

5.56 [5] <§5.7>  **For More Practice:** Microcode

5.57 [30] <§5.8> Using the strategy you developed in Exercise 5.53, modify the MIPS microinstruction format described in  Figure 5.7.1 and provide the complete microprogram for the `bcmp` instruction. Describe in detail how you extended the microcode so as to support the creation of more complex control structures (such as a loop) within the microcode. Has support for the `bcmp` instruction changed the size of the microcode? Will other instructions besides `bcmp` be affected by the change in the microinstruction format?

5.58 [5] <§5.8> A and B are registers defined through the following Verilog initialization code:

```
reg A,B
initial begin
    A = 1;
    B = 2;
end
```


Analyze the following two segments of Verilog description lines, and compare the results of variables A and B, and the operation done in each example.


```
a)      always @(negedge clock) begin
        A = B;
        B = A;
      end
b)      always @(negedge clock) begin
        A <= B;
        B <= A;
      end
```

5.59 [15] <§§5.4, 5.8> Write the ALUControl module in combinational Verilog using the following form as the basis:

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
    input ALUOp[1:0], FuncCode[5:0];
    output ALUCtl[3:0];
    ....
endmodule
```

5.60 [1 week] <§§5.3, 5.4, 5.8> Using a hardware simulation language such as Verilog, implement a functional simulator for the single-cycle version. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

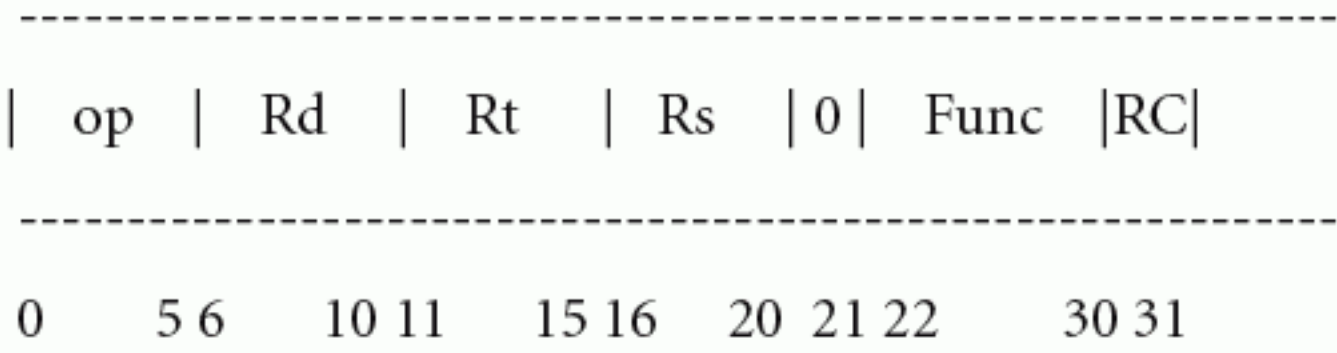
5.61 [2–4 hours] <§§4.7, 5.5, 5.8, 5.8> Extend the multicycle Verilog description in  5.8 by adding an implementation of the unsigned MIPS multiply instruction; assume it is implemented using the MIPS ALU and a shift and add operation.

5.62 [2–4 hours] <§§4.7, 5.5, 5.8, 5.9> Extend the multicycle Verilog description in  5.8 by adding an implementation of the unsigned MIPS divide instruction; assume it is implemented using the MIPS ALU with a one-bit-at-a-time algorithm.

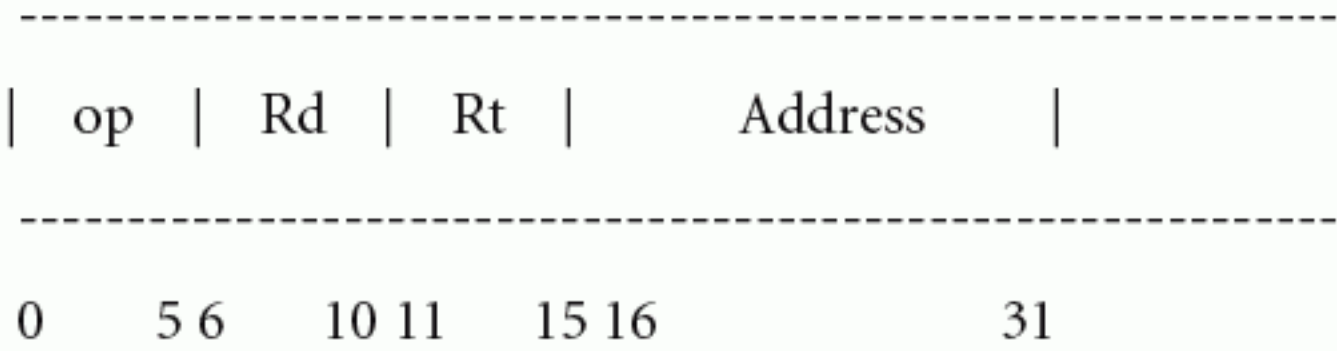
5.63 [1 week] <§§5.5, 5.8> Using a hardware simulation language such as Verilog, implement a functional simulator for a multicycle implementation of the design of a PowerPC processor. Build your simulator using an existing library of parts, if such a library is available. If the parts contain timing information, determine what the cycle time of your implementation will be.

Like MIPS, the PowerPC instructions are 32 bits each. Assume that your instruction set supports the following instruction formats:

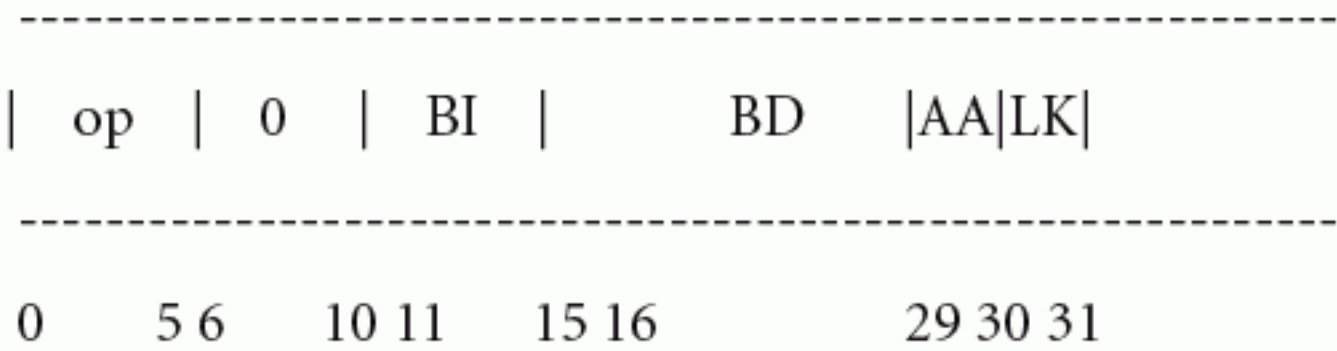
R-type



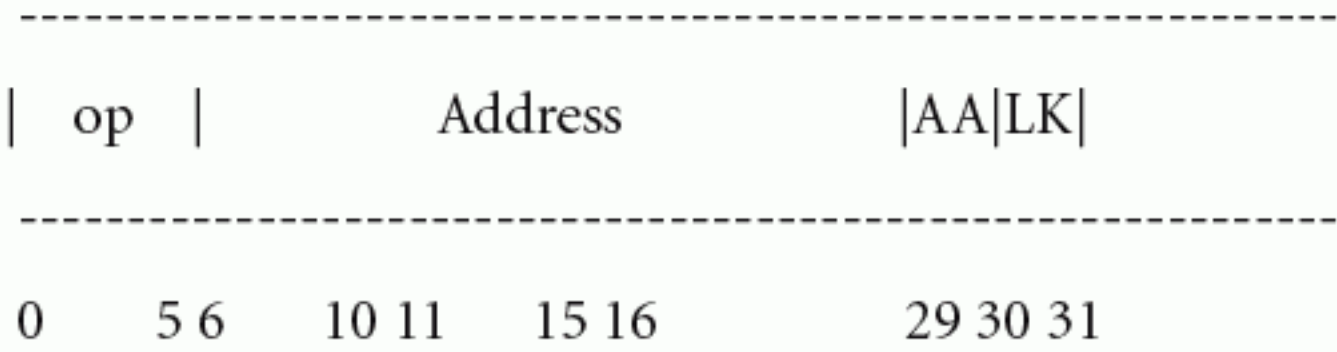
Load/store & immediate



Branch conditional



Jump



RC-reg



0 1 2 3

Func field (22:30): Similar to MIPS, identifies function code.

RC bit(31): IF set (1), update the RC-reg control bits to reflect the results of the instruction (all R-type).

AA(30): 1 indicates that the given address is an absolute address;
0 indicates relative address.

LK: IF 1, updates LNKR (the link register), which can be later used for subroutine return implementation.

BI: Encodes the branch condition (e.g., beg -> BI = 2, blt -> BI = 0, etc.)

BD: Branch relative destination.

Your simplified PowerPC implementation should be able to implement the following instructions:

add: add \$Rd, \$Rt, \$Rs (\$Rd <- \$Rt + \$Rs)
 addi \$Rd, \$Rt, #n (\$Rd <- \$Rt + #n)

subtract: sub \$Rd, \$Rt, \$Rs (\$Rd <- \$Rt - \$Rs)
 subi \$Rd, \$Rt, #n (\$Rd <- \$Rt - #n)

load: lw \$Rd, Addr(\$Rt) (\$Rd <- Memory[\$Rt + Addr])

store: sw \$Rd, Addr(\$Rt) (Memory[\$Rt + Addr] <- \$Rd)

AND, OR: and/or \$Rd, \$Rt, \$Rs (\$Rd <- \$Rt AND/OR \$Rs)
 andi/ori \$Rd, \$Rt, #n (\$Rd <- \$Rt AND/OR #n)

Jump: jmp Addr (PC <- Addr)

Branch conditional: Beq Addr (CR[2]==1? PC<- PC+BD : PC <- PC+4)

subroutine call: jal Addr (LNKR <- PC+4; PC<- Addr)

subroutine restore: Ret (PC <- LNKR)

5.64 [Discussion] <§§5.7, 5.10, 5.11> Hypothesis: If the first implementation of an architecture uses microprogramming, it affects the instruction set architecture.

implementation do you think the architect had in mind when designing the instruction set architecture?

5.65 [Discussion] <§§5.7, 5.12> Wilkes invented microprogramming in large part to simplify construction of control. Since 1980, there has been an explosion of computer-aided design software whose goal is also to simplify construction of control. This has made control design much easier. Can you find evidence, based either on the tools or on real designs, that supports or refutes this hypothesis?

5.66 [Discussion] <§5.12> The MIPS instructions and the MIPS microinstructions have many similarities. What would make it difficult for a compiler to produce MIPS microcode rather than macrocode? What changes to the microarchitecture would make the microcode more useful for this application.

Answers to Check Yourself

§5.1, page 289: 3.

§5.2, page 292: false.

§5.3, page 299: A.

§5.4, page 318: Yes, MemtoReg and RegDst are inverses of one another. Yes, simply use the other signal and flip the order of the inputs to the multiplexor!

§5.5, page 340: 1. False. 2. Maybe: If the signal PCSource[0] is always set to zero when it is a don't care (which is most states), then it is identical to PCWriteCond.

§5.6, page 346: No, since the value of 11, which was formerly unused, is now used!

§5.7, page 5.7-13: 4 tables with 55 entries (don't forget the primary dispatch!)

§5.8, page 5.8-7: 1. 0, 1, 1, X, 0. 2. No, since state is not assigned on every path.

Computers in the Real World

Empowering the Disabled

Problem to solve: To overcome the obstacles faced by disabled people.

Solution: Use robotics, sensors, and computer control to replace or supplement damaged limbs and organs.

The picture on the right shows a system developed for a firefighter who was injured while fighting a fire. Sensors in latex fingers instantly register hot and cold, and an electronic interface in his artificial limb stimulates the nerve endings in his upper arm, which then pass the information to his brain. The \$3000 system allows his hand to feel pressure and weight, so for the first time since losing his arms in a 1986 accident, he can pick up a can of soda without crushing it or having it slip through his fingers. The main enabling device is an electronic interface that can transmit signals to nerve endings in Whitten's upper arm, which then pass this information to his brain.

Harvey Fishman and Mark Peterman of Stanford have taken steps towards information technology that might someday treat age-related blindness. Their approach is to



Firefighter Ken Whitten proudly displays his new bionic arm.

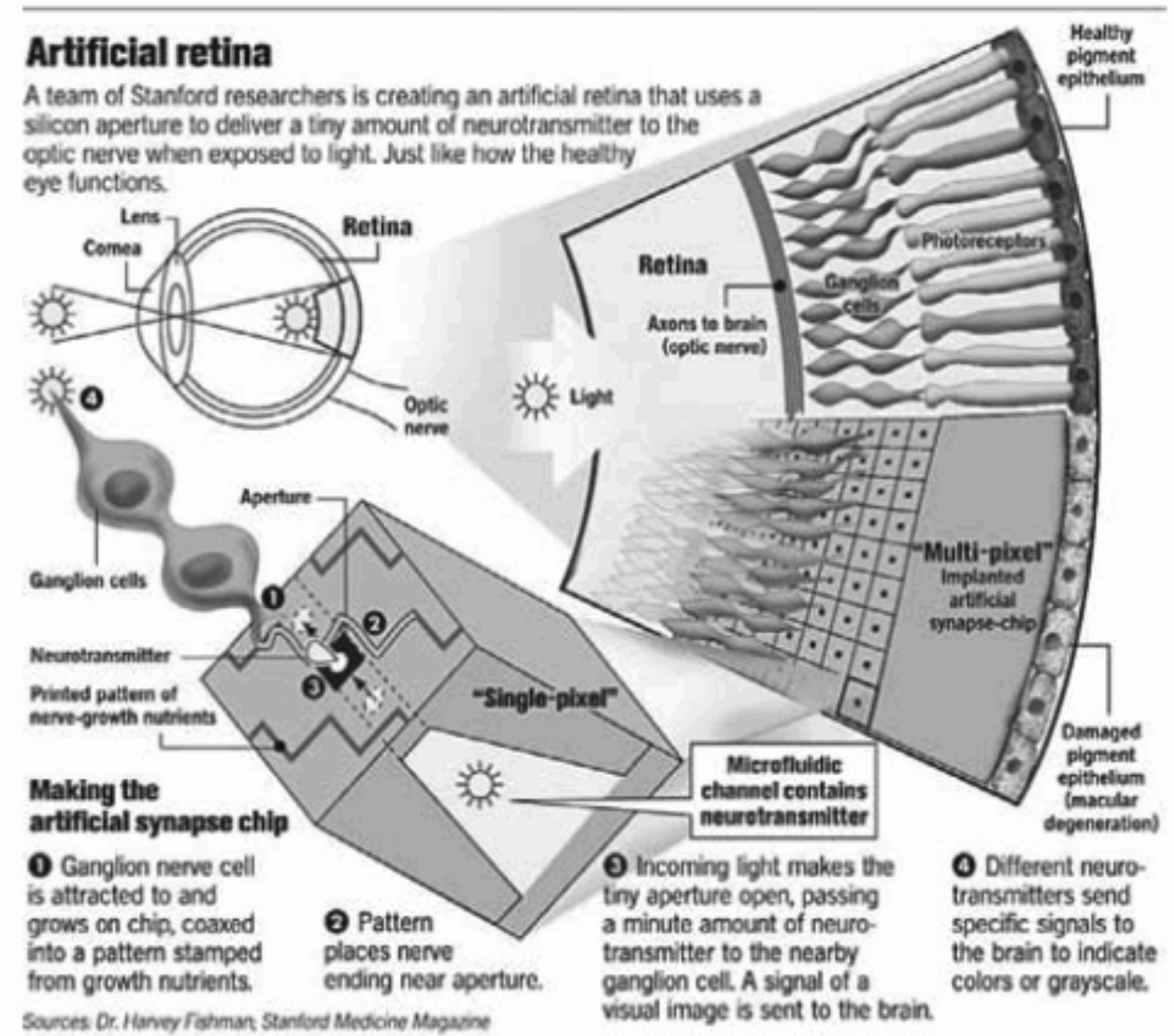
bypass the photoreceptors of the eye with a signal from a digital camera that connects directly to the visual system. They are developing a neural interface to the visual system called the artificial synapse chip. The challenge is to turn electrical signals into the chemicals that cells use to communicate. This chip is attached to cells, and from the cell's perspective the artificial synapse is simply a hole in the silicon. This hole is connected to a reservoir of neurotransmitter. When an electric field is applied to the chip, the neurotransmitter is pumped through the hole, stimulating nearby cells. In 2003 they have

created four artificial synapses on a chip one centimeter on a side.

Although this work is in its early stages, the potential is not limited to eye problems. According to Fishman, "Anywhere there's a severing of a nerve connection, there's a potential for us to reconnect it."

To learn more see these references on the  library

- Rick Smolan and Jennifer Erwit, *One Digital Day: How the Microchip Is Changing Our World*, Times Publishing, 1998
- Peterman et al, "The artificial synapse chip: A flexible retinal interface based on directed retinal cell growth and neurotransmitter stimulation," *Artificial Organs*: 27(11), November 18, 2003



Artificial retina using artificial synapse chips. From *The San Francisco Chronicle*, January 5, 2004.



6

Enhancing Performance with Pipelining

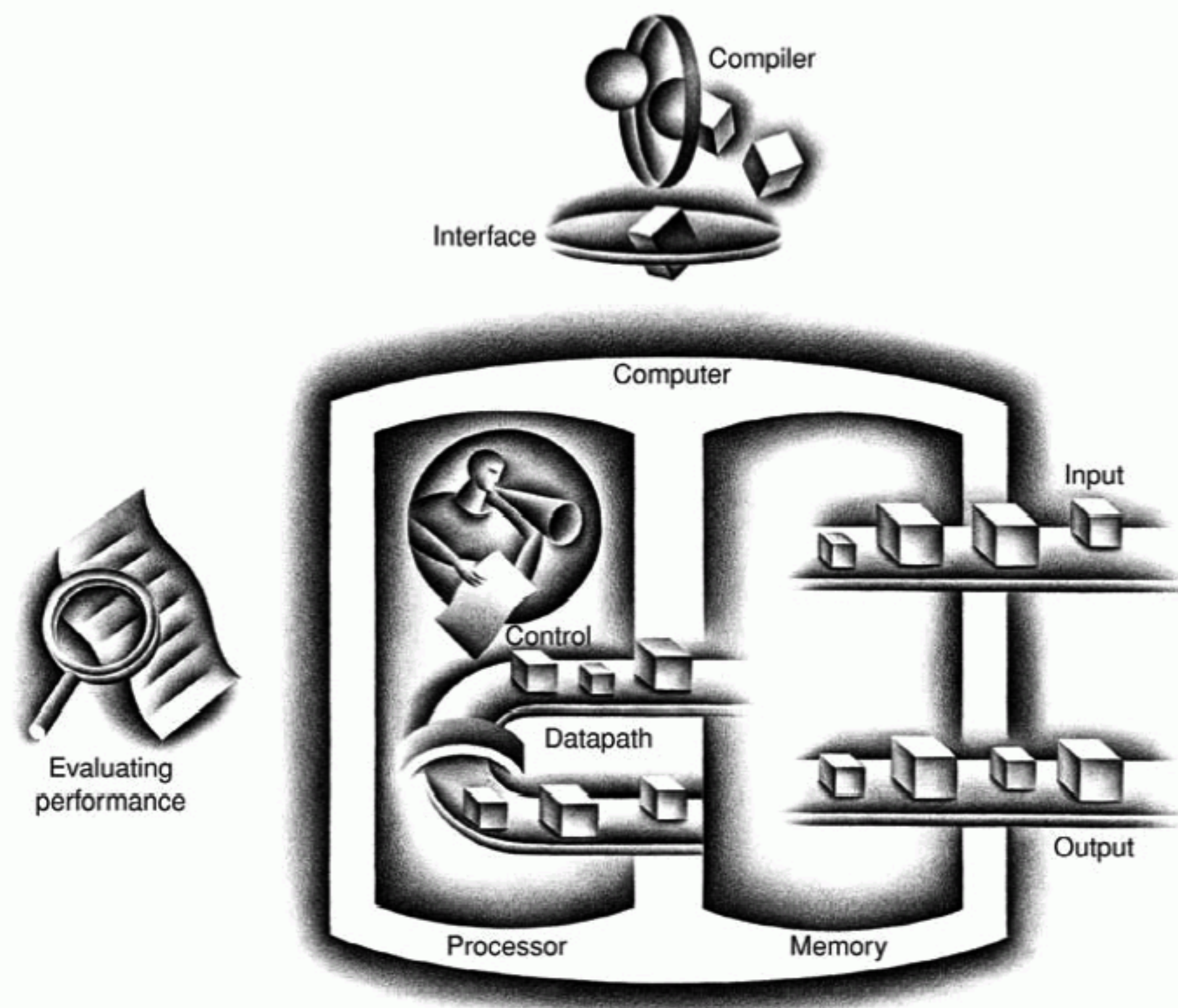
*Thus times do shift,
each thing his turn does hold;
New things succeed,
as former things grow old.*

Robert Herrick

Hesperides: Ceremonies for Christmas Eve, 1648

6.1	An Overview of Pipelining	370
6.2	A Pipelined Datapath	384
6.3	Pipelined Control	399
6.4	Data Hazards and Forwarding	402
6.5	Data Hazards and Stalls	413
6.6	Branch Hazards	416
 6.7	Using a Hardware Description Language to Describe and Model a Pipeline	426
6.8	Exceptions	427
6.9	Advanced Pipelining: Extracting More Performance	432
6.10	Real Stuff: The Pentium 4 Pipeline	448
6.11	Fallacies and Pitfalls	451
6.12	Concluding Remarks	452
 6.13	Historical Perspective and Further Reading	454
6.14	Exercises	454

The Five Classic Components of a Computer



Never waste time.

American proverb

pipelining An implementation technique in which multiple instructions are overlapped in execution, much like to an assembly line.

6.1

An Overview of Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processors fast.

This section relies heavily on one analogy to give an overview of the pipelining terms and issues. If you are interested in just the big picture, you should concentrate on this section and then skip to Sections 6.9 and 6.10 to see an introduction to the advanced pipelining techniques used in recent processors such as the Pentium III and 4. If you are interested in exploring the anatomy of a pipelined computer, this section is a good introduction to Sections 6.2 through 6.8.

Anyone who has done a lot of laundry has intuitively used pipelining. The *non-pipelined* approach to laundry would be

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, then start over with the next dirty load.

The *pipelined* approach takes much less time, as Figure 6.1 shows. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and the next dirty load into the washer. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called *stages* in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.

The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour. Pipelining improves throughput of our laundry system without improving the time to complete a single load. Hence, pipelining would not decrease the time to complete one load of laundry, but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.

If all the stages take about the same amount of time and there is enough work to do, then the speedup due to pipelining is equal to the number of stages in the pipeline, in this case four: washing, drying, folding, and putting away. So, pipelined laundry is potentially four times faster than nonpipelined: 20 loads would

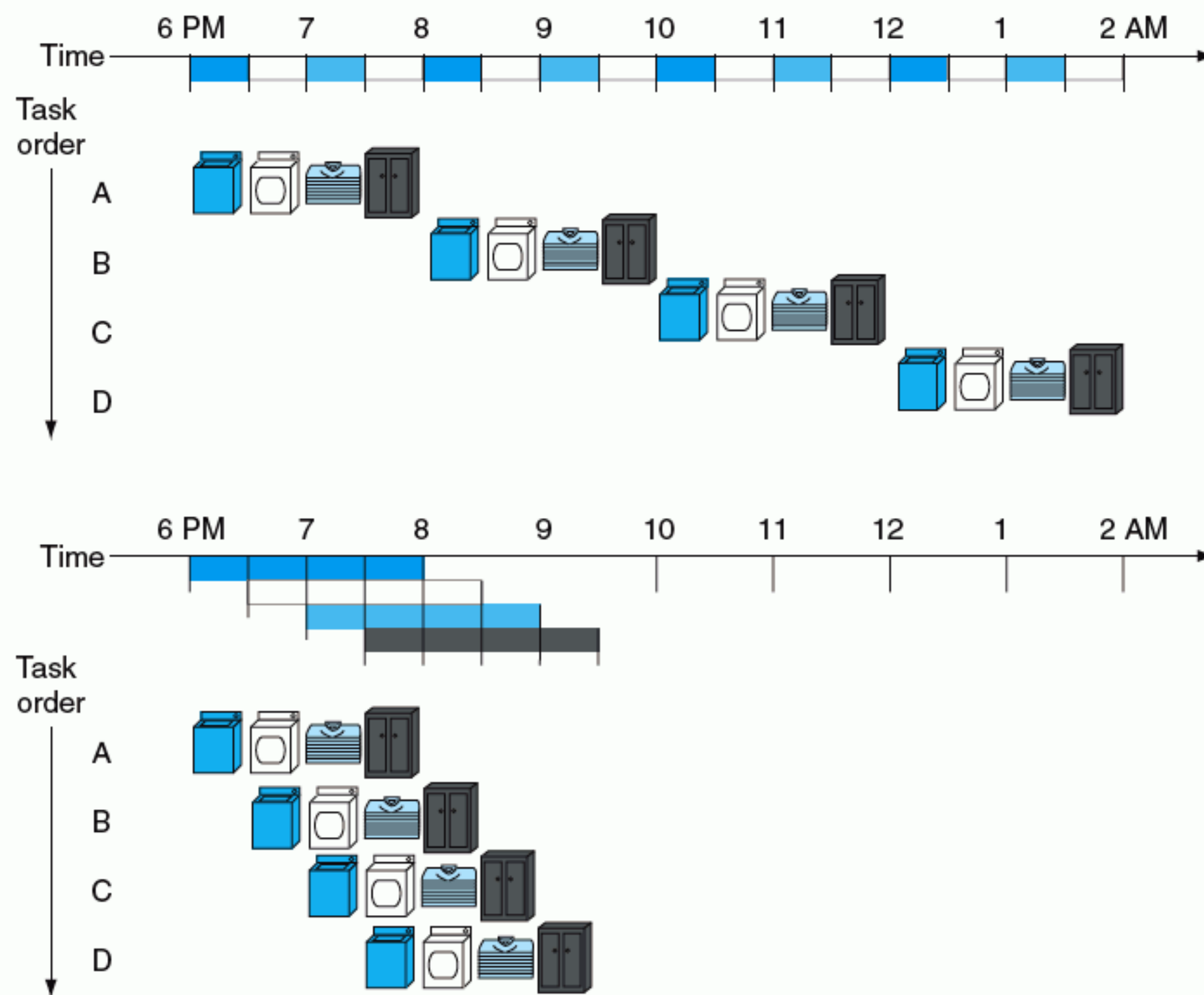


FIGURE 6.1 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, “folder,” and “storer” each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It’s only 2.3 times faster in Figure 6.1 because we only show 4 loads. Notice that at the beginning and end of the workload in the pipelined version in Figure 6.1, the pipeline is not completely full, this start-up and wind-down affects performance when the number of tasks is not large compared to the number of stages in the pipeline. If the number of loads is much larger than 4, then the stages will be full most of the time and the increase in throughput will be very close to 4.

The same principles apply to processors where we pipeline instruction execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously.

3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

Hence, the MIPS pipeline we explore in this chapter has five stages. The following example shows that pipelining speeds up instruction execution just as it speeds up the laundry.

EXAMPLE

Single-Cycle versus Pipelined Performance

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to eight instructions: load word (lw), store word (sw), add (add), subtract (sub), and (and), or (or), set-less-than (slt), and branch-on-equal (beq).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write. As we said in Chapter 5, in the single-cycle model every instruction takes exactly 1 clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

Figure 6.2 shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction—in Figure 6.2 it is lw—so the time required for every instruction is 800 ps. Similarly to Figure 6.1, Figure 6.3 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3×800 ns or 2400 ps.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×200 ps or 600 ps.

ANSWER

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 6.2 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

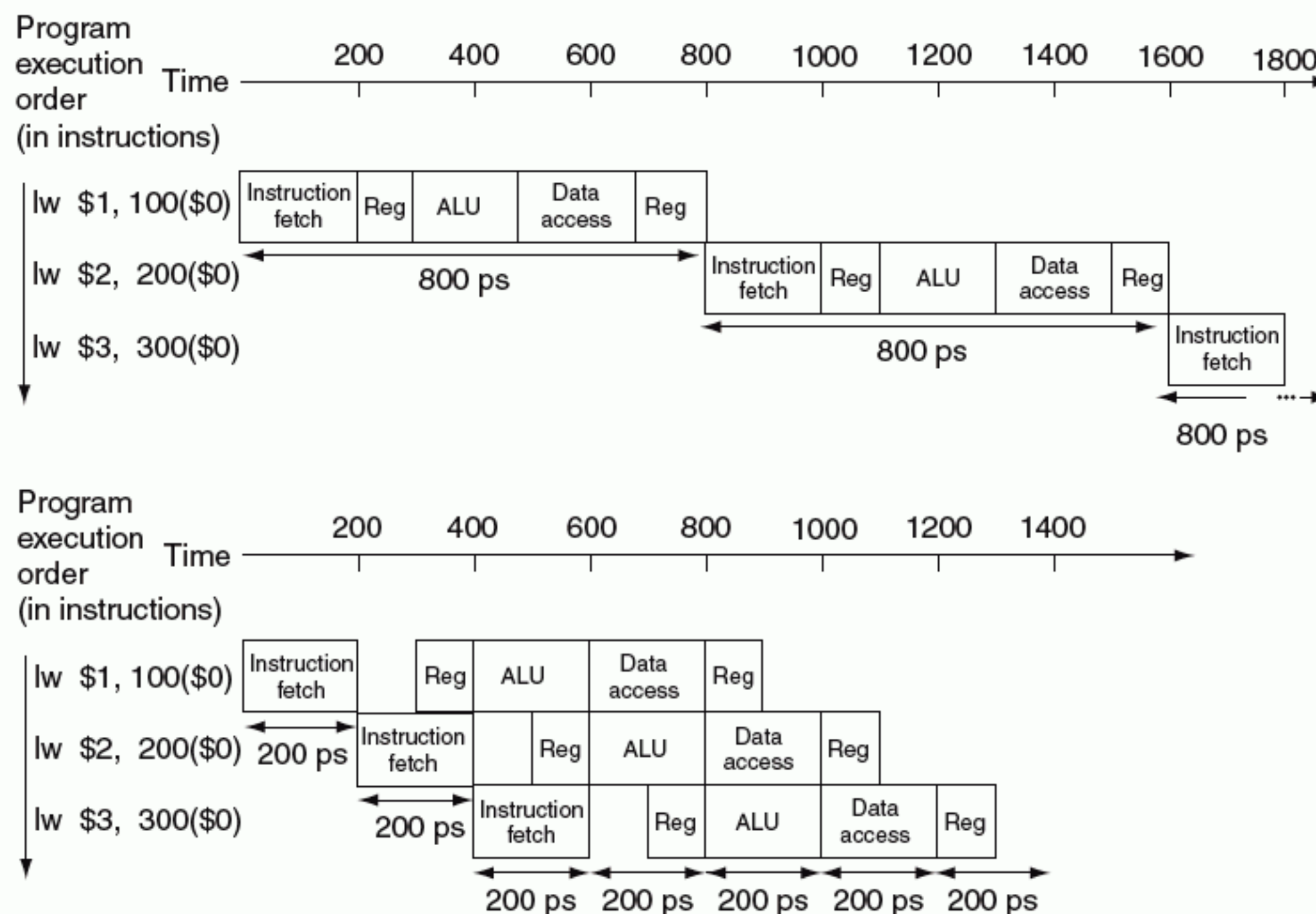


FIGURE 6.3 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 6.2. In this case we see a fourfold speedup on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 6.1. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The computer pipeline stage times are limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

We can turn the pipelining speedup discussion above into a formula. If the stages are perfectly balanced, then the time between instructions on the pipelined processor—assuming ideal conditions—is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Under ideal conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle. The example shows, however, that the stages may be imperfectly balanced. In addition, pipelining involves some overhead, the source of which will be more clear shortly. Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speedup will be less than the number of pipeline stages.

Moreover, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 1400 ps versus 2400 ps. Of course, this is because the number of instructions is not large. What would happen if we increased the number of instructions? We could extend the previous figures to 1,000,003 instructions. We would add 1,000,000 instructions in the pipelined example; each instruction adds 200 ps to the total execution time. The total execution time would be $1,000,000 \times 200 \text{ ps} + 1400 \text{ ps}$, or 200,001,400 ps. In the nonpipelined example, we would add 1,000,000 instructions, each taking 800 ps, so total execution time would be $1,000,000 \times 800 \text{ ps} + 2400 \text{ ps}$, or 800,002,400 ps. Under these ideal conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

$$\frac{800,002,400 \text{ ps}}{200,001,400 \text{ ps}} \approx 4.00 \approx \frac{800 \text{ ps}}{200 \text{ ps}}$$

Pipelining improves performance by *increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction*, but instruction throughput is the important metric because real programs execute billions of instructions.

Designing Instruction Sets for Pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the MIPS instruction set, which was designed for pipelined execution.

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the

second stage. In an instruction set like the IA-32, where instructions vary from 1 byte to 17 bytes, pipelining is considerably more challenging. As we saw in Chapter 5, all recent implementations of the IA-32 architecture actually translate IA-32 instructions into simple microoperations that look like MIPS instructions. As we will see in Section 6.10, the Pentium 4 actually pipelines the microoperations rather than the native IA-32 instructions!

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages. We will shortly see the downside of longer pipelines.

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage. If we could operate on the operands in memory, as in the IA-32, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, as discussed in Chapter 2, operands must be aligned in memory. Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

Structural Hazards

The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 6.3 had a fourth instruction, we would see that in the same clock cycle that the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

structural hazard An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.

data hazard Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

forwarding Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

EXAMPLE

ANSWER

Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads that have completed drying and are ready to fold and those that have finished washing and are ready to dry must wait.

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (\$s0):

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to add three bubbles to the pipeline.

Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

Forwarding with Two Instructions

For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in Figure 6.4 to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in Figure 6.1.

Figure 6.5 shows the connection to forward the value in \$s0 after the execution stage of the add instruction as input to the execution stage of the sub instruction.

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first

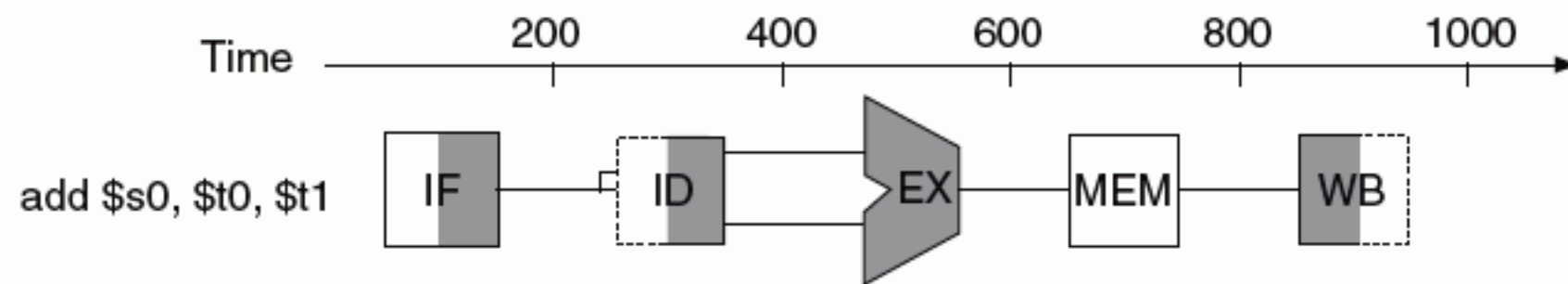


FIGURE 6.4 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 6.1 on page 371. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, *MEM* has a white background because *add* does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of *ID* is shaded in the second stage because the register file is read, and the left half of *WB* is shaded in the fifth stage because the register file is written.

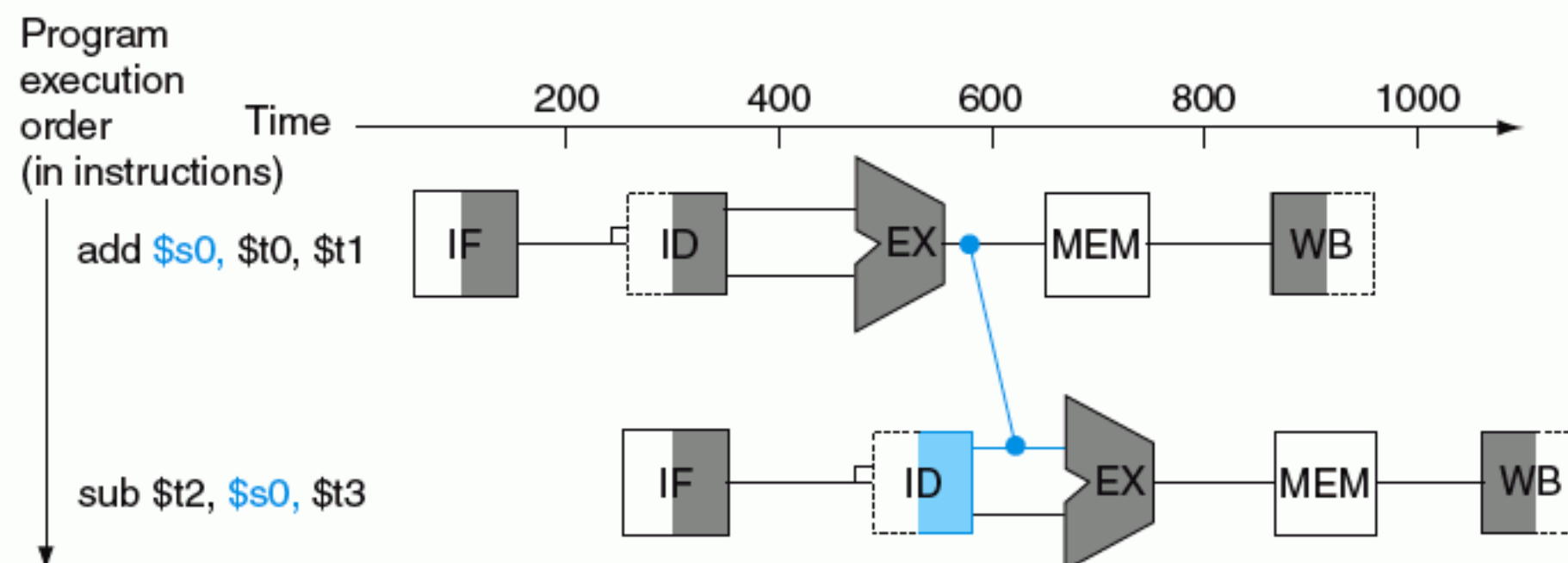


FIGURE 6.5 Graphical representation of forwarding. The connection shows the forwarding path from the output of the *EX* stage of *add* to the input of the *EX* stage for *sub*, replacing the value from register *\$s0* read in the second stage of *sub*.

instruction to the input of the execution stage of the following, since that would mean going backwards in time.

Forwarding works very well and is described in detail in Section 6.4. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction were a load of *\$s0* instead of an *add*. As we can imagine from looking at Figure 6.5, the desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of *sub*. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**, as Figure 6.6 shows. This figure shows an important pipeline concept, officially called a **pipeline stall**, but often given the nickname *bubble*. We shall see stalls elsewhere in the pipeline. Section 6.5 shows how we can handle hard cases like these, using either hardware detection and stalls or software that treats the load delay like a branch delay.

load-use data hazard A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.

pipeline stall Also called bubble. A stall initiated in order to resolve a hazard.

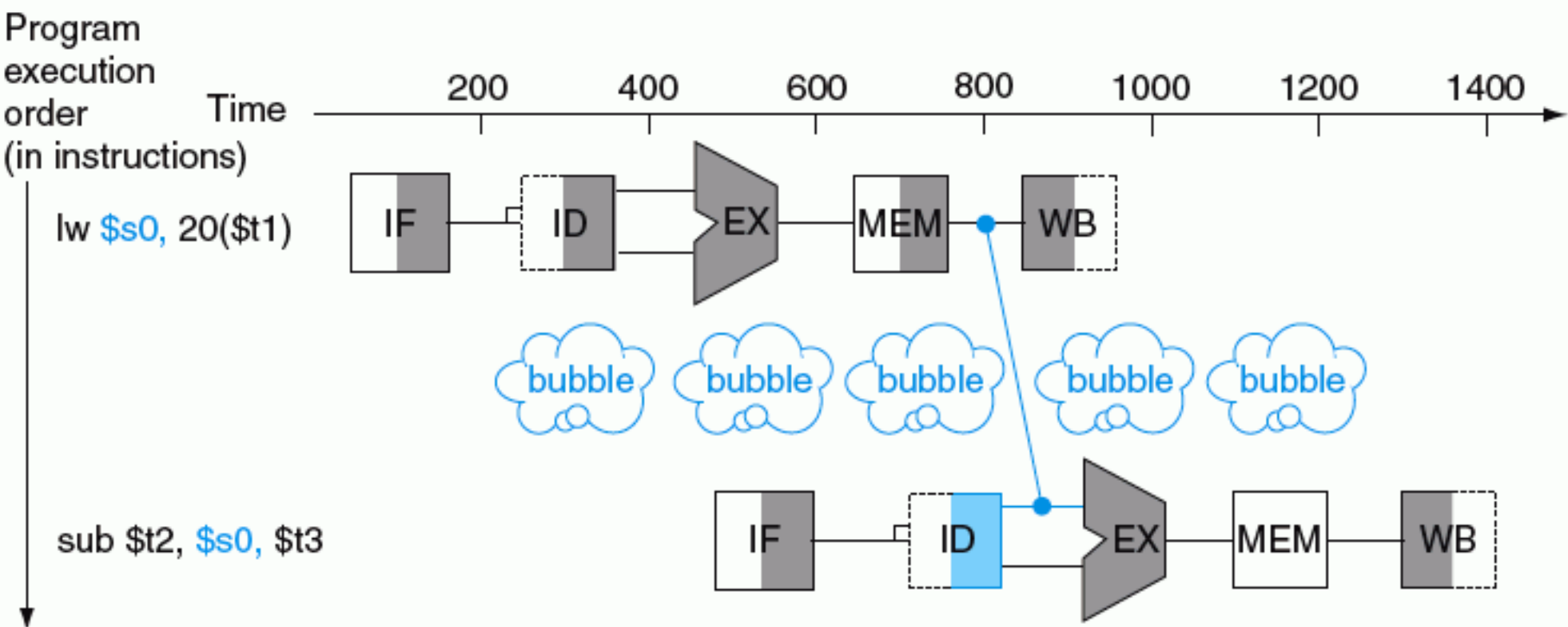


FIGURE 6.6 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 6.5 shows the details of what really happens in the case of a hazard.

EXAMPLE

Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
A = B + E;  
C = B + F;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from \$t0:

```
lw    $t1, 0($t0)  
lw    $t2, 4($t0)  
add   $t3, $t1, $t2  
sw    $t3, 12($t0)  
lw    $t4, 8($t0)  
add   $t5, $t1, $t4  
sw    $t5, 16($t0)
```

Find the hazards in the following code segment and reorder the instructions to avoid any pipeline stalls.

Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction. Notice that bypassing eliminates several other potential hazards including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction eliminates both hazards:

```
lw      $t1, 0($t0)
lw      $t2, 4($t1)
lw      $t4, 8($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

Forwarding yields another insight into the MIPS architecture, in addition to the four mentioned on page 374–375. Each MIPS instruction writes at most one result and does so near the end of the pipeline. Forwarding is harder if there are multiple results to forward per instruction or they need to write a result early on in instruction execution.

Elaboration: The name “forwarding” comes from the idea that the result is passed forward from an earlier instruction to a later instruction. “Bypassing” comes from passing the result by the register file to the desired unit.

Control Hazards

The third type of hazard is called a **control hazard**, arising from the need to make a decision based on the results of one instruction while others are executing.

Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here is the first of two solutions to control hazards in the laundry room and its computer equivalent.

Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula. This conservative option certainly works, but it is slow.

ANSWER

control hazard Also called **branch hazard**. An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

The equivalent decision task in a computer is the branch instruction. Notice that we must begin fetching the instruction following the branch on the very next clock cycle. But, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory! Just as with laundry, one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

Let’s assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline (see Section 6.6 for details). Even with this extra hardware, the pipeline involving conditional branches would look like Figure 6.7. The 1w instruction, executed if the branch fails, is stalled one extra 200-ps clock cycle before starting.

EXAMPLE

ANSWER

Performance of “Stall on Branch”

Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.

Figure 3.26 on page 228 in Chapter 3 shows that branches are 13% of the instructions executed in SPECint2000. Since other instructions run have a CPI of 1 and branches took one extra clock cycle for the stall, then we would see a CPI of 1.13 and hence a slowdown of 1.13 versus the ideal case. Notice that this includes only branches and that jumps might also incur a stall.

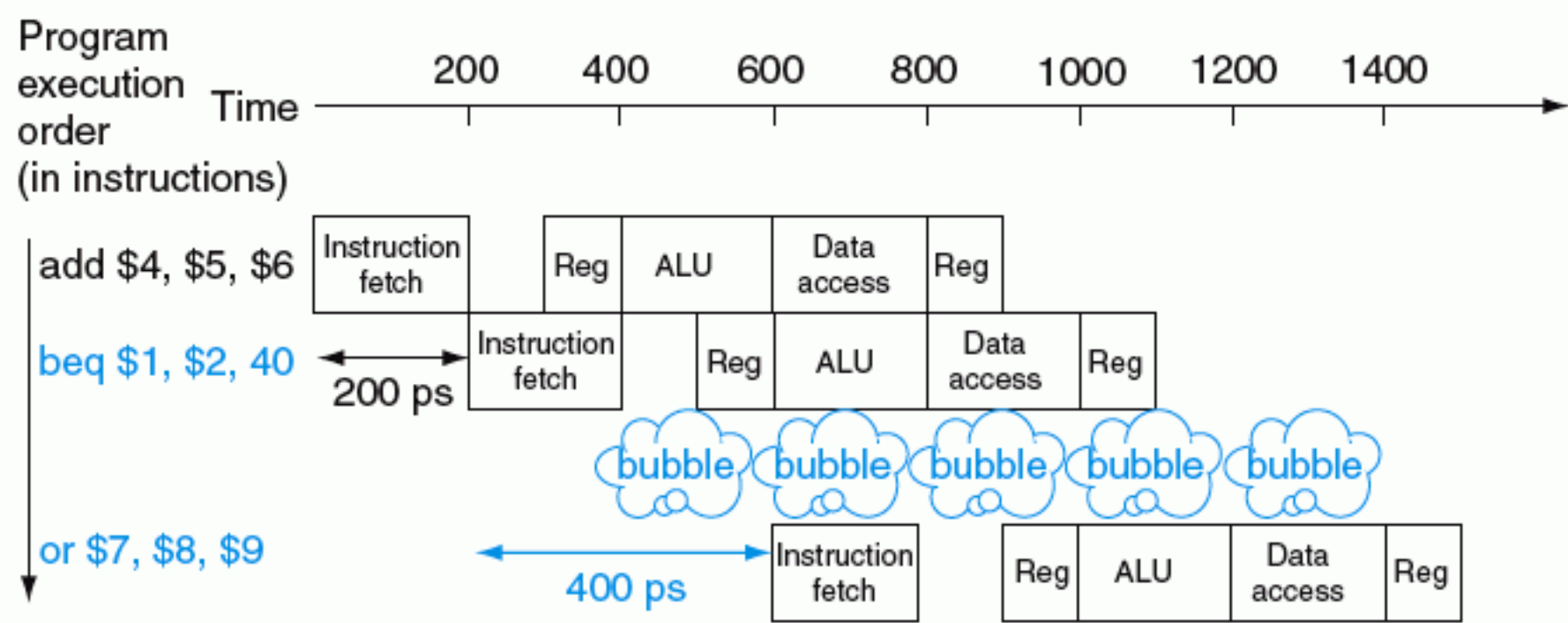


FIGURE 6.7 Pipeline showing stalling on every conditional branch as solution to control hazards. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 6.6. The effect on performance, however, is the same as would occur if a bubble were inserted.

If we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we'd see an even larger slowdown if we stall on branches. The cost of this option is too high for most computers to use and motivates a second solution to the control hazard:

Predict: If you're pretty sure you have the right formula to wash uniforms, then just predict that it will work and wash the second load while waiting for the first load to dry. This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Computers do indeed use *prediction* to handle branches. One simple approach is to always predict that branches will be **untaken**. When you're right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall. Figure 6.8 shows such an example.

untaken branch One that falls through to the successive instruction. A taken branch is one that causes transfer to the branch target.

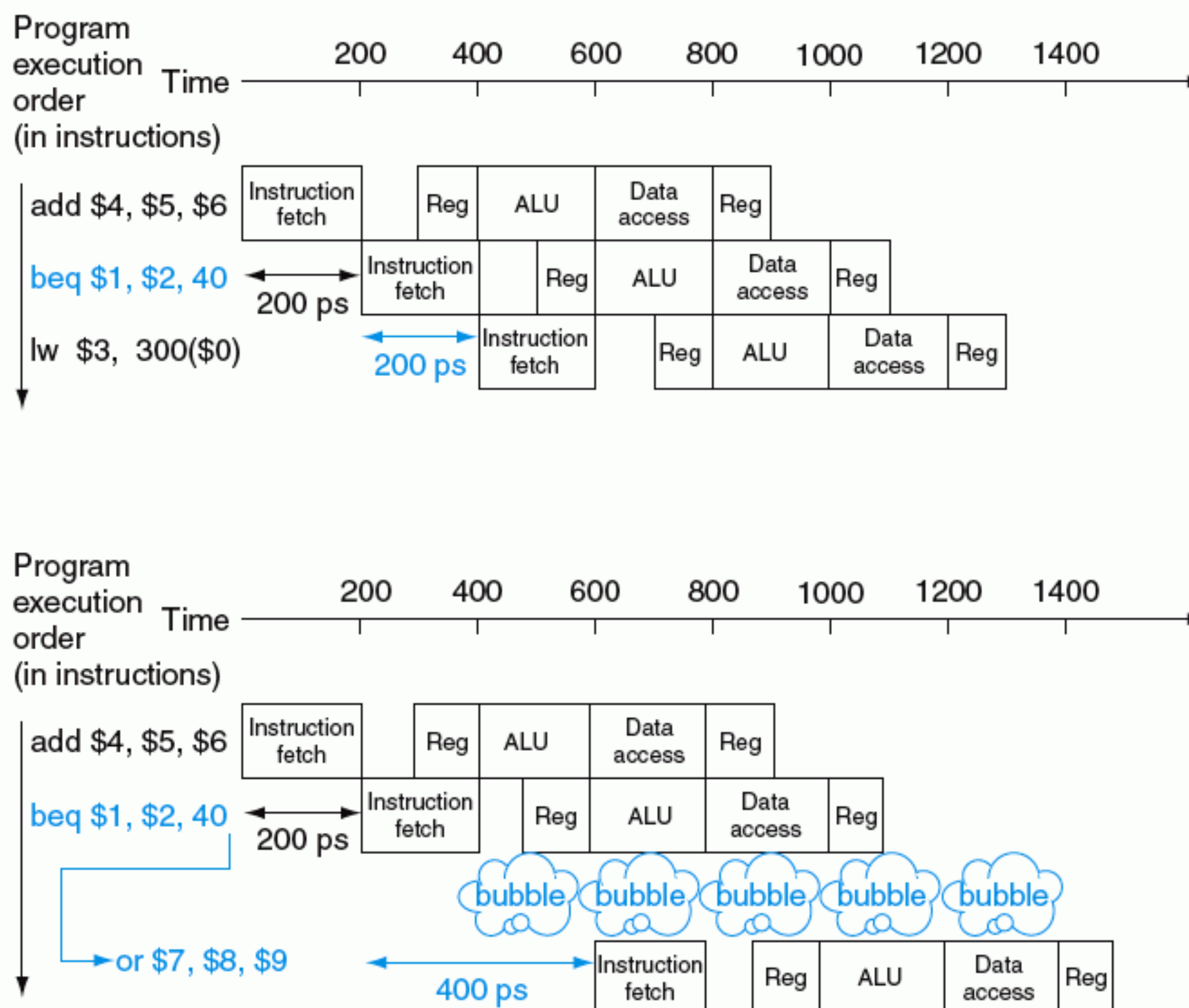


FIGURE 6.8 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 6.7, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 6.6 will reveal the details.

branch prediction A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

A more sophisticated version of **branch prediction** would have some branches predicted as taken and some as untaken. In our analogy, the dark or home uniforms might take one formula while the light or road uniforms might take another. As a computer example, at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backwards, we could always predict taken for branches that jump to an earlier address.


Such rigid approaches to branch prediction rely on stereotypical behavior and don't account for the individuality of a specific branch instruction. *Dynamic* hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next guess depending on the success of recent guesses. One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future. As we will see later, the amount and type of history kept have become extensive with the result being that dynamic branch predictors can correctly predict branches with over 90% accuracy (see Section 6.6). When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address. In our laundry analogy, we must stop taking new loads so that we can restart the load that we incorrectly predicted.

As in the case of all other solutions to control hazards, longer pipelines exacerbate the problem, in this case by raising the cost of misprediction. Solutions to control hazards are described in more detail in Section 6.6.

Elaboration: There is a third approach to the control hazard, called *delayed decision*. In our analogy, whenever you are going to make such a decision about laundry, just place a load of nonfootball clothes in the washer while waiting for football uniforms to dry. As long as you have enough dirty clothes that are not affected by the test, this solution works fine.

Called the *delayed branch* in computers, this is the solution actually used by the MIPS architecture. The delayed branch always executes the next sequential instruction, with the branch taking place *after* that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer. MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that *follows* this safe instruction. In our example, the add instruction before the branch in Figure 6.7 does not affect the branch and can be moved after the branch to fully hide the branch delay. Since delayed branches are useful when the branches are short, no processor uses a delayed branch of more than 1 cycle. For longer branch delays, hardware-based branch prediction is usually used.

Pipeline Overview Summary

Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see  Chapter 9), it is fundamentally invisible to the programmer.

In the next sections of this chapter, we cover the concept of pipelining using the MIPS instruction subset `lw`, `sw`, `add`, `sub`, `and`, `or`, `sllt`, and `beq` (same as Chapter 5) and a simplified version of its pipeline. We then look at the problems that pipelining introduces and the performance attainable under typical situations.

If you wish to focus more on the software and the performance implications of pipelining, you now have sufficient background to skip to Section 6.9. Section 6.9 introduces advanced pipelining concepts, such as superscalar and dynamic scheduling, and Section 6.10 examines the pipeline of the Pentium 4 microprocessor.

Alternatively, if you are interested in understanding how pipelining is implemented and the challenges of dealing with hazards, you can proceed to examine the design of a pipelined datapath, explained in Section 6.2, and the basic control, explained in Section 6.3. You can then use this understanding to explore the implementation of forwarding in Section 6.4, and the implementation of stalls in Section 6.5. You can then read Section 6.6 to learn more about solutions to branch hazards, and then see how exceptions are handled in Section 6.8.

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction, also called the **latency**. For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in Chapter 4, pipelining improves instruction *throughput* rather than individual instruction *execution time* or *latency*.

Instruction sets can either simplify or make life harder for pipeline designers, who must already cope with structural, control, and data hazards. Branch prediction, forwarding, and stalls help make a computer fast while still getting the right answers.

The BIG Picture

latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.

Understanding Program Performance

Outside of the memory system, the effective operation of the pipeline is usually the most important factor in determining the CPI of the processor and hence its performance. As we will see in Section 6.9, understanding the performance of a modern multiple-issue pipelined processor is complex and requires understanding more than just the issues that arise in a simple pipelined processor. Nonetheless, structural, data, and control hazards remain important in both simple pipelines and in more sophisticated ones.

For modern pipelines, structural hazards usually revolve around the floating-point unit, which may not be fully pipelined, while control hazards are usually more of a problem in integer programs, which tend to have higher branch frequencies as well as less predictable branches. Data hazards can be performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower branch frequency and more regular access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular access involving more use of pointers. As we will see in Section 6.9, there are more ambitious compiler and hardware techniques for reducing data dependences through scheduling.

Check Yourself

For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding:

Sequence 1	Sequence 2	Sequence 3
lw \$t0,0(\$t0) add \$t1,\$t0,\$t0	add \$t1,\$t0,\$t0 addi \$t2,\$t0,#5 addi \$t4,\$t1,#5	addi \$t1,\$t0,#1 addi \$t2,\$t0,#2 addi \$t3,\$t0,#2 addi \$t3,\$t0,#4 addi \$t5,\$t0,#5

There is less in this than meets the eye.
Tallulah Bankhead, remark to Alexander Wollcott, 1922

6.2 A Pipelined Datapath

Figure 6.9 shows the single-cycle datapath from Chapter 5. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we

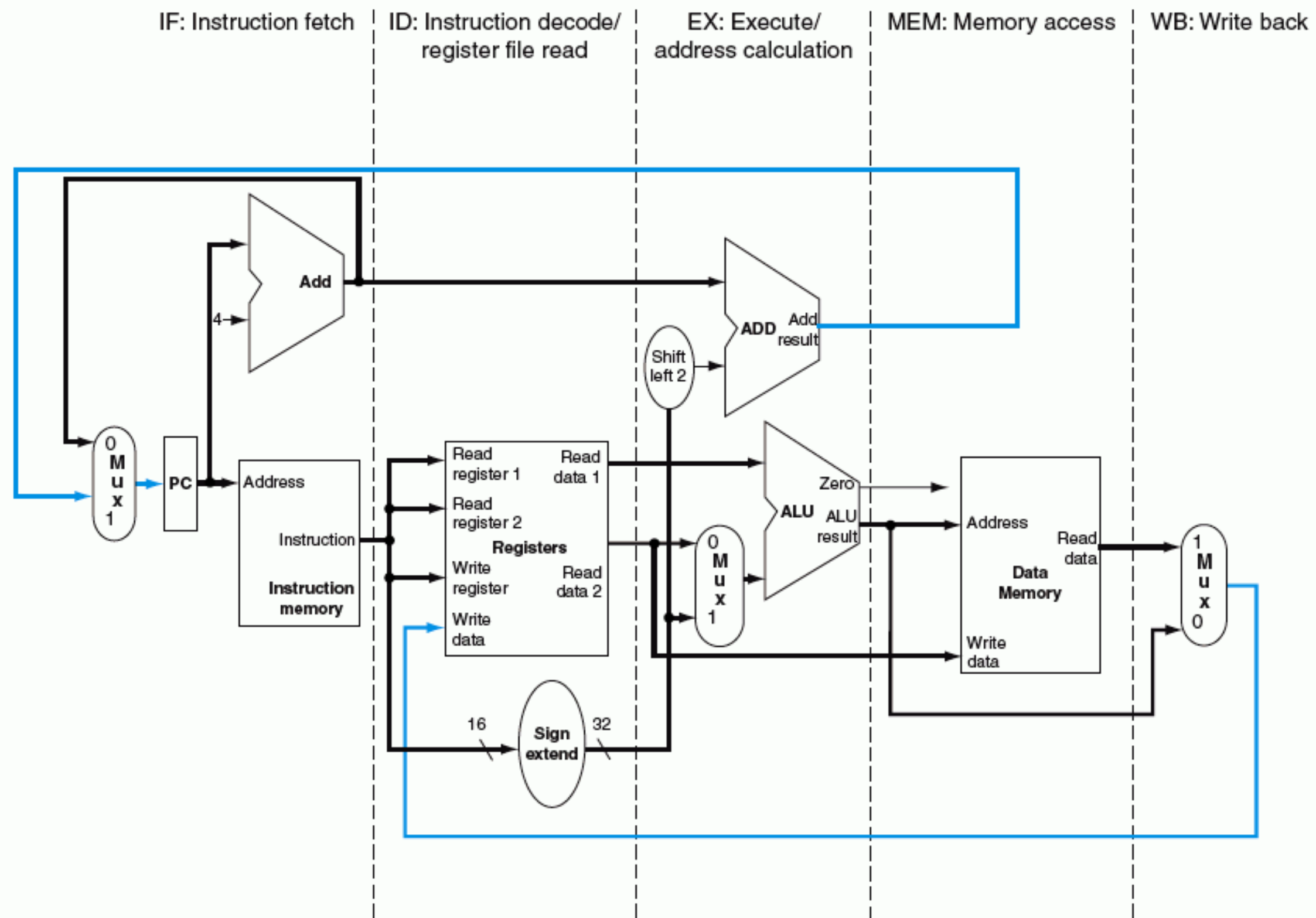


FIGURE 6.9 The single-cycle datapath from Chapter 5 (similar to Figure 5.17 on page 307). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

In Figure 6.9, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution. Going back to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backwards.

There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

Data flowing from right to left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements. Note that the first right-to-left arrow can lead to data hazards and the second leads to control hazards.

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a time line to show their relationship. Figure 6.10 shows the execution of the instructions in Figure 6.3 by displaying their private datapaths on a common time line. We use a stylized version of the datapath in Figure 6.9 to show the relationships in Figure 6.10.

Figure 6.10 seems to suggest that three instructions need three datapaths. In Chapter 5, we added registers to hold data so that portions of the datapath could be shared during instruction execution; we use the same technique here to share the multiple datapaths. For example, as Figure 6.10 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by other instructions during the other four stages.

To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages in Figure 6.9. This change is similar to the registers added in Chapter 5 when we went from a single-cycle to a multicycle datapath. Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

Figure 6.11 shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor—the register file, memory, or the PC—so a separate pipeline register is redundant to the state that is updated. For example, a load instruction will place its result in 1 of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.

Of course, every instruction updates the PC, whether by incrementing it or by setting it to a branch destination address. The PC can be thought of as a pipeline register: one that feeds the IF stage of the pipeline. Unlike the shaded pipeline reg-

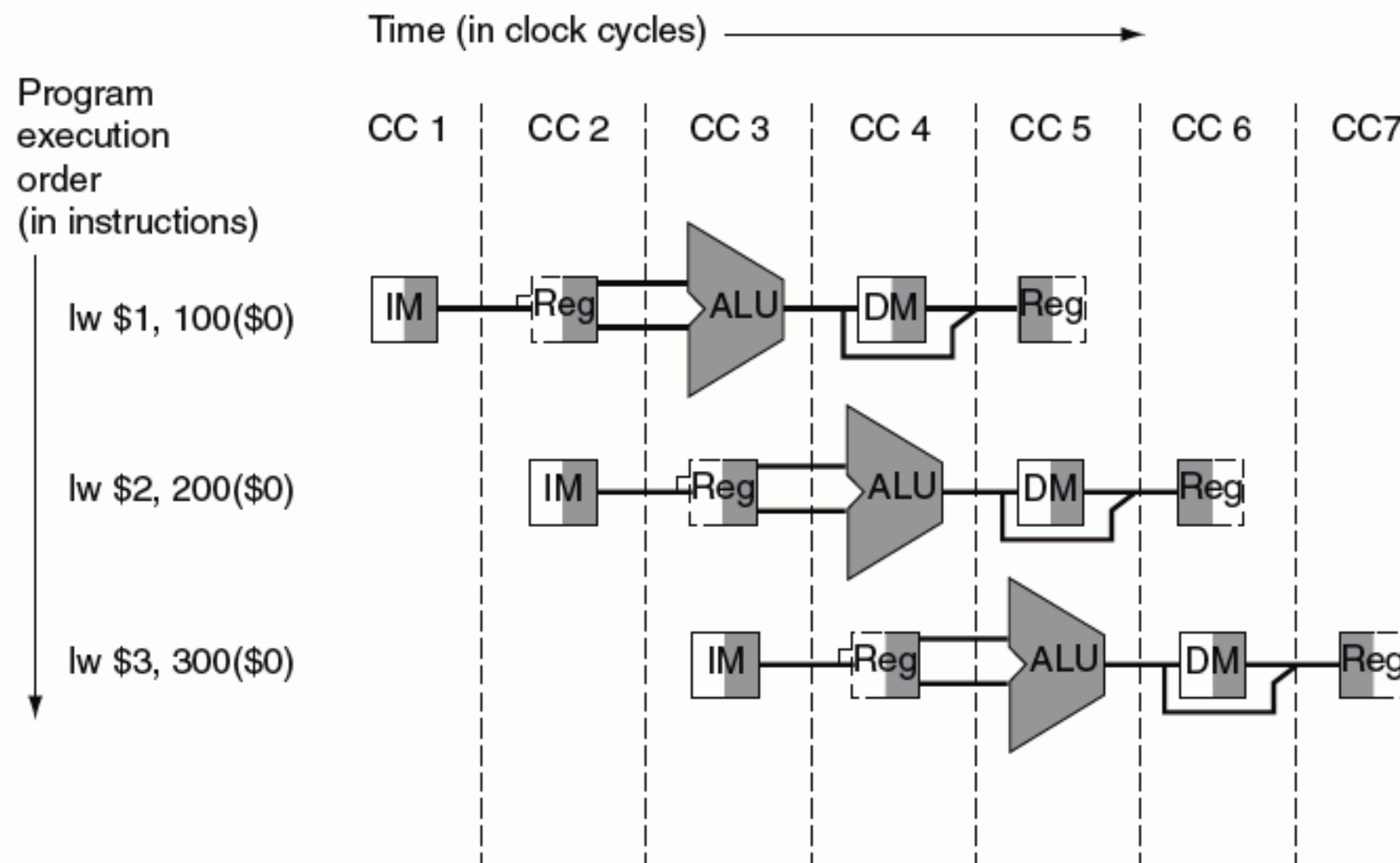


FIGURE 6.10 Instructions being executed using the single-cycle datapath in Figure 6.9, assuming pipelined execution. Similar to Figures 6.4 through 6.6, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 6.9. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

isters in Figure 6.11, however, the PC is part of the visible architectural state; its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded. In the laundry analogy, you could think of the PC as corresponding to the basket that holds the load of dirty clothes before the wash step!

To show how the pipelining works, throughout this chapter we show sequences of figures to demonstrate operation over time. These extra pages would seem to require much more time for you to understand. Fear not; the sequences take much less time than it might appear because you can compare them to see what changes occur in each clock cycle. Sections 6.4 and 6.5 describe what happens when there are data hazards between pipelined instructions; ignore them for now.

Figures 6.12 through 6.14, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined

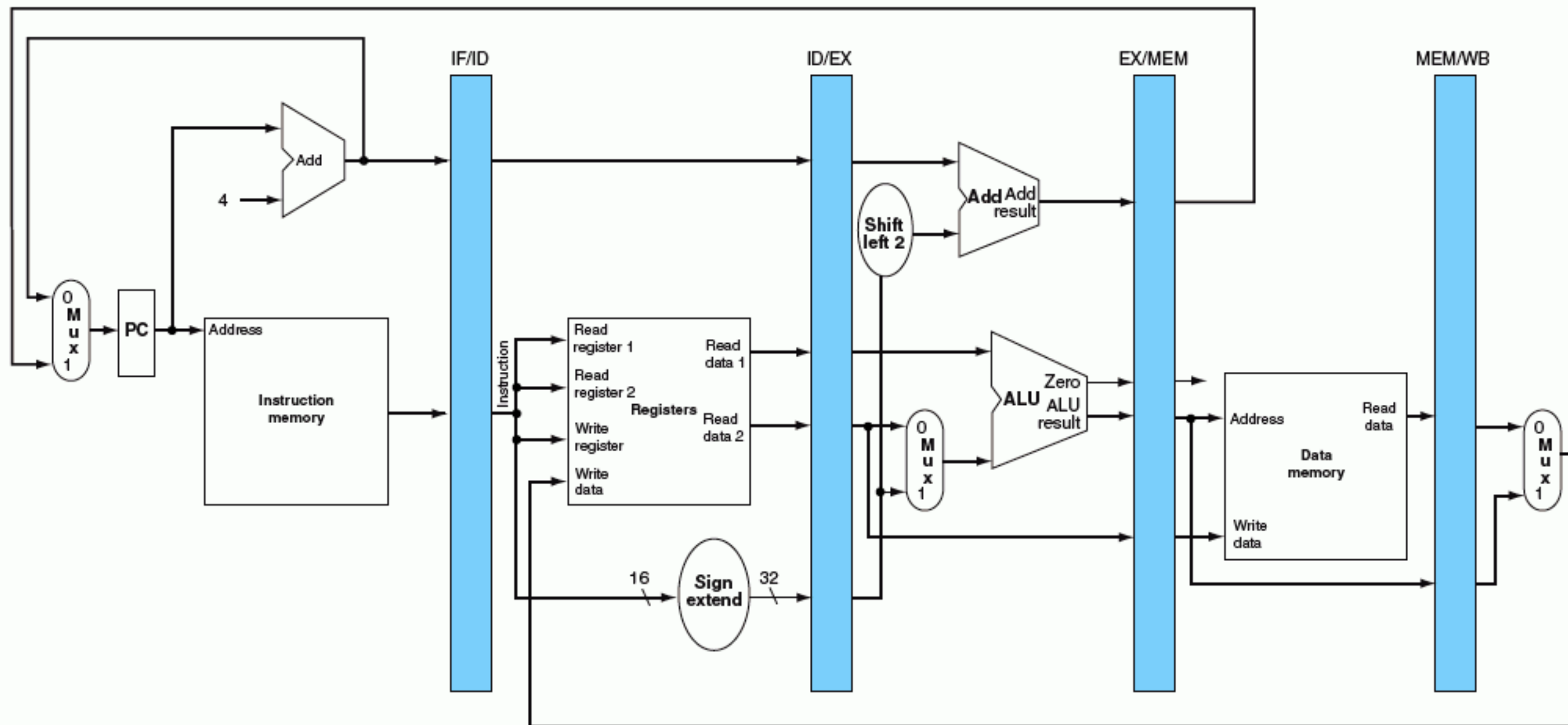


FIGURE 6.11 The pipelined version of the datapath in Figure 6.9. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

execution. We show a load first because it is active in all five stages. As in Figures 6.4 through 6.11, we highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*. We show the instruction abbreviation *lw* with the name of the pipe stage that is active in each figure. The five stages are the following:

1. *Instruction fetch*: The top portion of Figure 6.12 shows the instruction being read from memory using the address in the PC and then placed in the IF/ID pipeline register. The IF/ID pipeline register is similar to the Instruction register in Figure 5.26 on page 320. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as *beq*. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

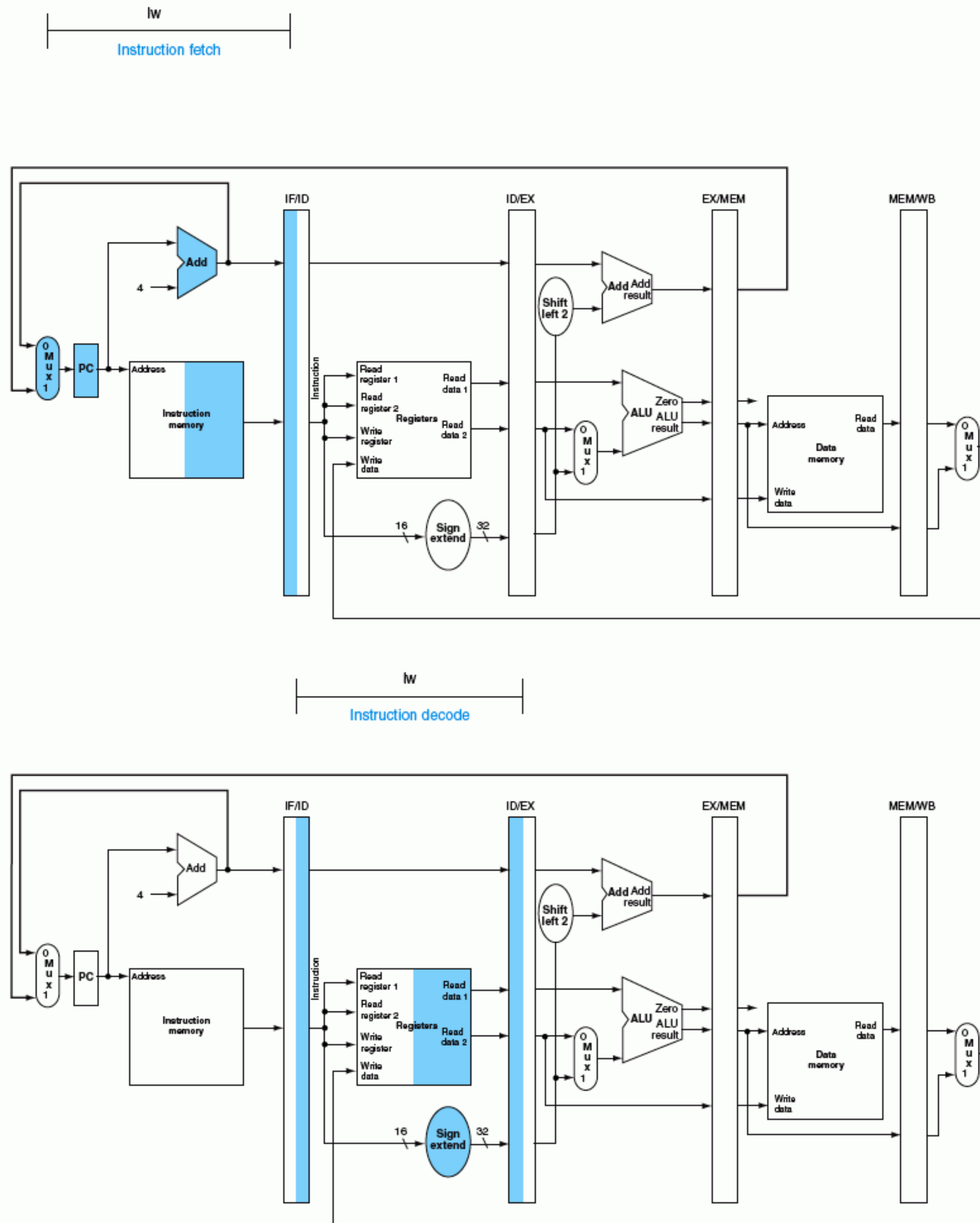


FIGURE 6.12 IF and ID: first and second pipe stages of an instruction, with the active portions of the datapath in Figure 6.11 highlighted. The highlighting convention is the same as that used in Figure 6.4. As in Chapter 5, there is no confusion when reading and writing registers because the contents change only on the clock edge. Although the load needs only the top register in stage 2, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register. We don't need all three operands, but it simplifies control to keep all three.

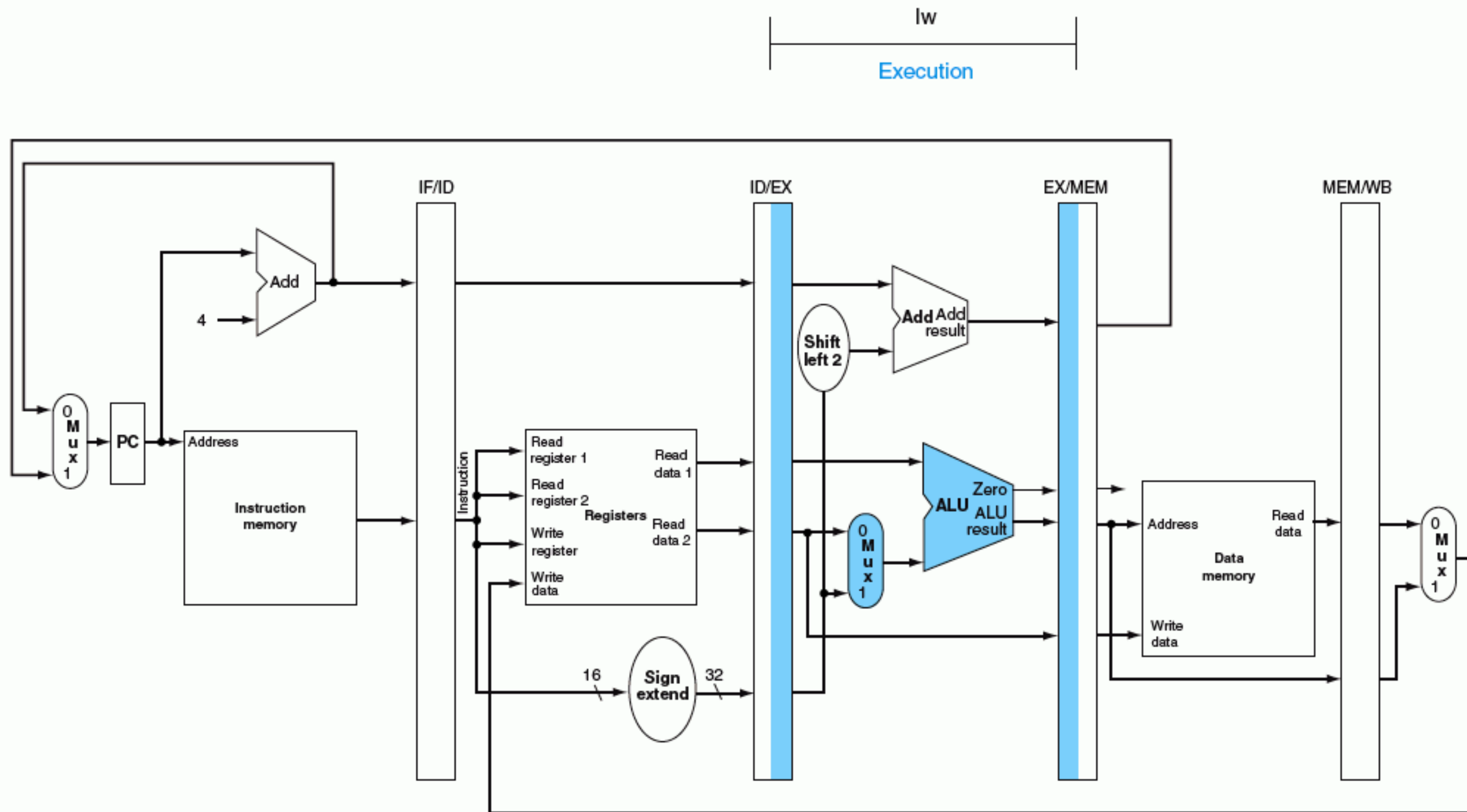


FIGURE 6.13 EX: the third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 6.11 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

2. *Instruction decode and register file read:* The bottom portion of Figure 6.12 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.
3. *Execute or address calculation:* Figure 6.13 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 6.14 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

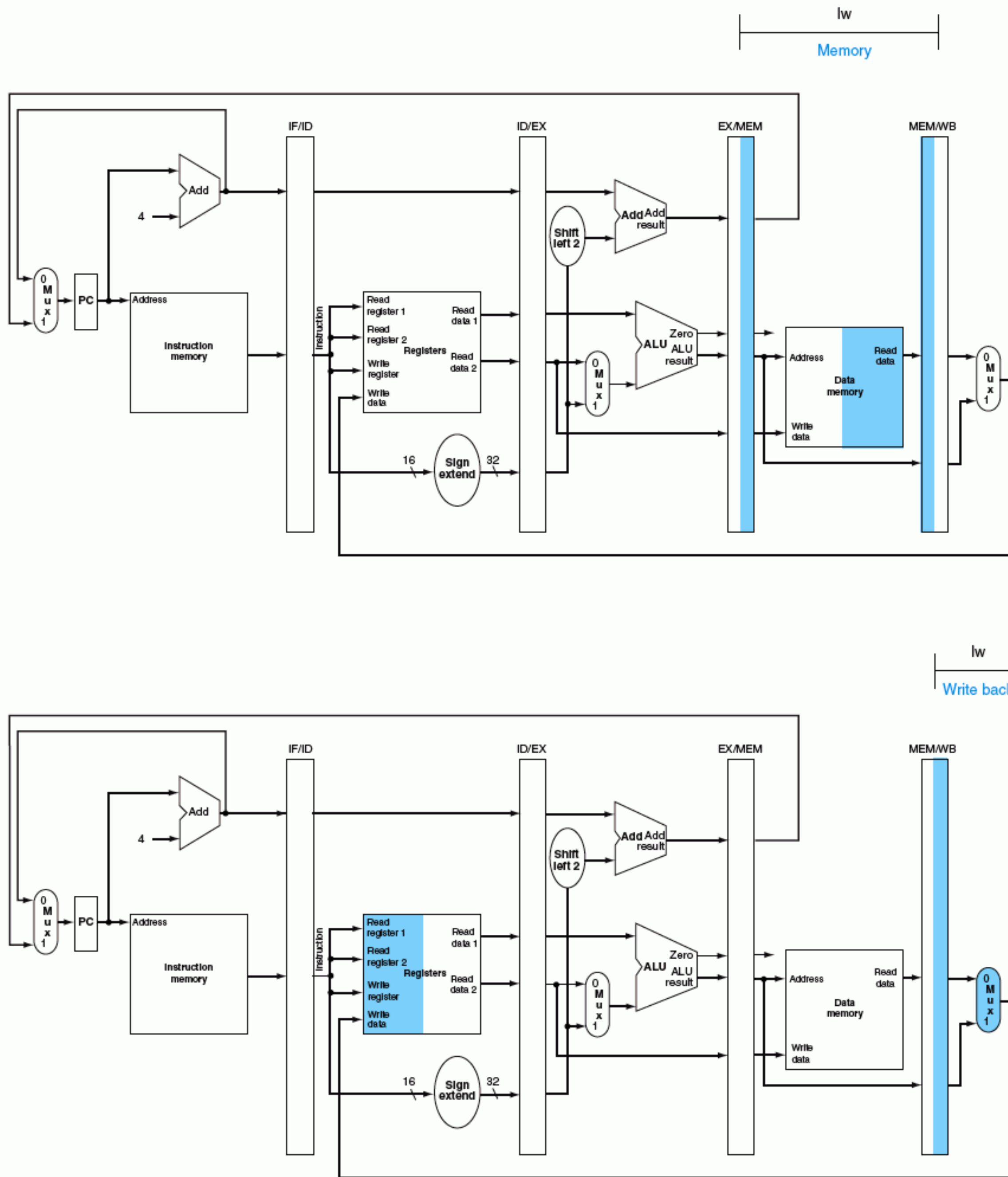


FIGURE 6.14 MEM and WB: the fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 6.11 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register. Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

5. *Write back:* The bottom portion of Figure 6.14 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register. Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are the five pipe stages of the store instruction:

1. *Instruction fetch:* The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of Figure 6.12 works for store as well as load.
2. *Instruction decode and register file read:* The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 6.12 for load instructions also shows the operations of the second stage for stores. These first two stages are executed by all instructions, since it is too early to know the type of the instruction.
3. *Execute and address calculation:* Figure 6.15 shows the third step; the effective address is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 6.16 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.
5. *Write back:* The bottom portion of Figure 6.16 shows the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do because later instructions are already progressing at the maximum rate.

The store instruction again illustrates that to pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise, the information is lost when the next instruction enters that pipeline stage. For the store instruction we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.

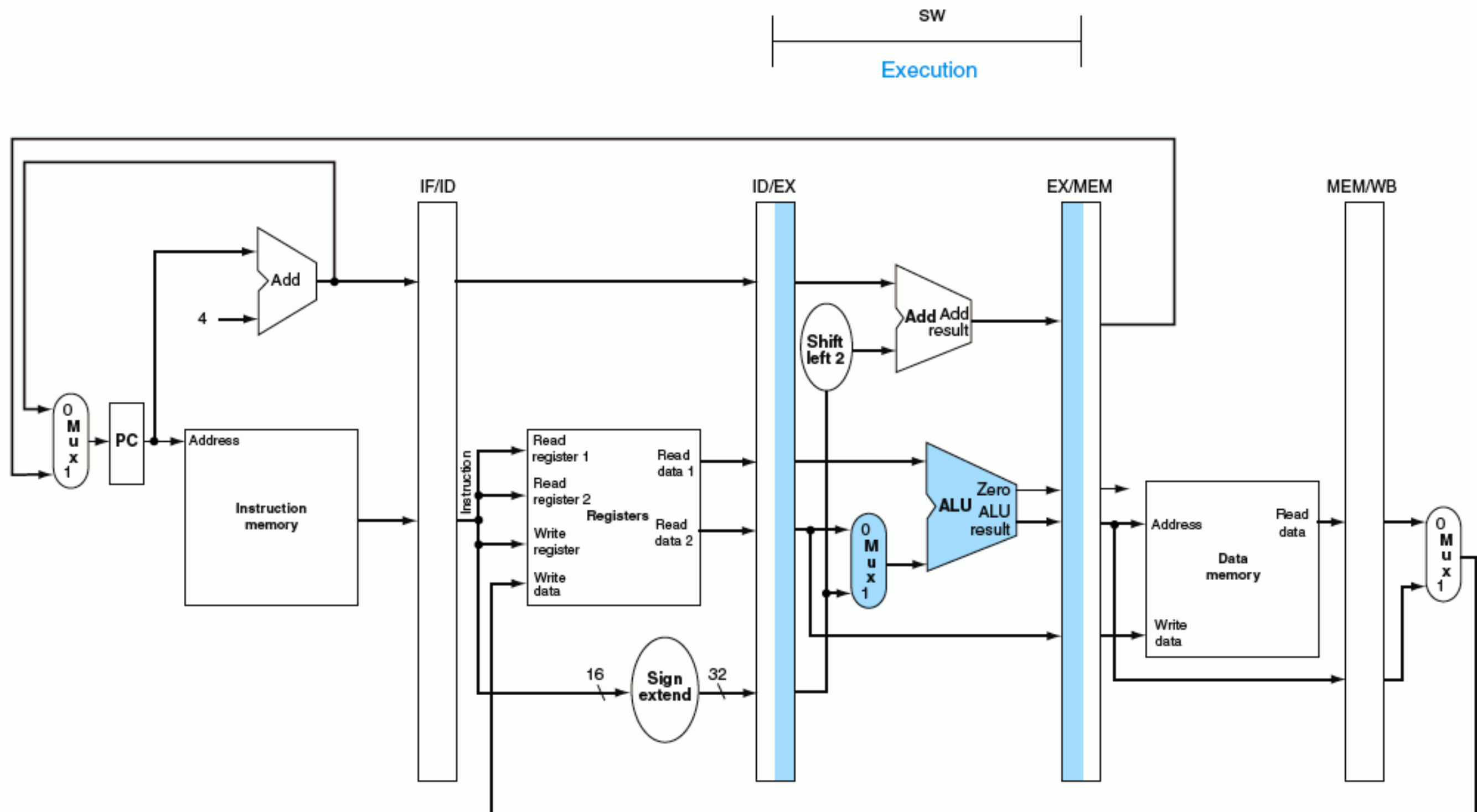


FIGURE 6.15 EX: the third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 6.13, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a *single* pipeline stage. Otherwise we would have a *structural hazard* (see page 375). Hence these components, and their control, can be associated with a single pipeline stage.

Now we can uncover a bug in the design of the load instruction. Did you see it? Which register is changed in the final stage of the load? More specifically, which instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably *after* the load instruction!

Hence, we need to preserve the destination register number in the load instruction. Just as store passed the register *contents* from the ID/EX to the EX/MEM pipeline registers for use in the MEM stage, load must pass the register *number* from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in

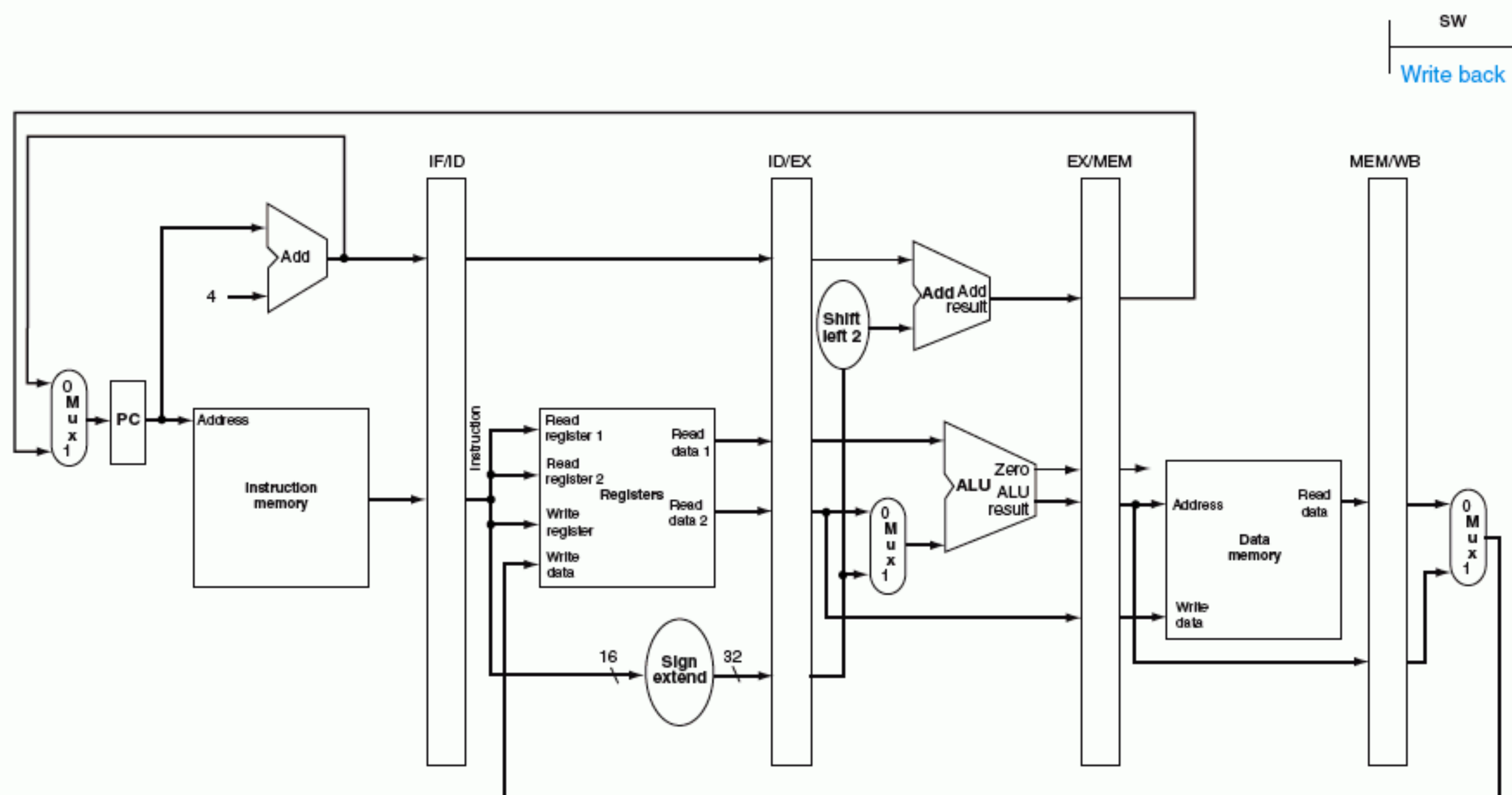
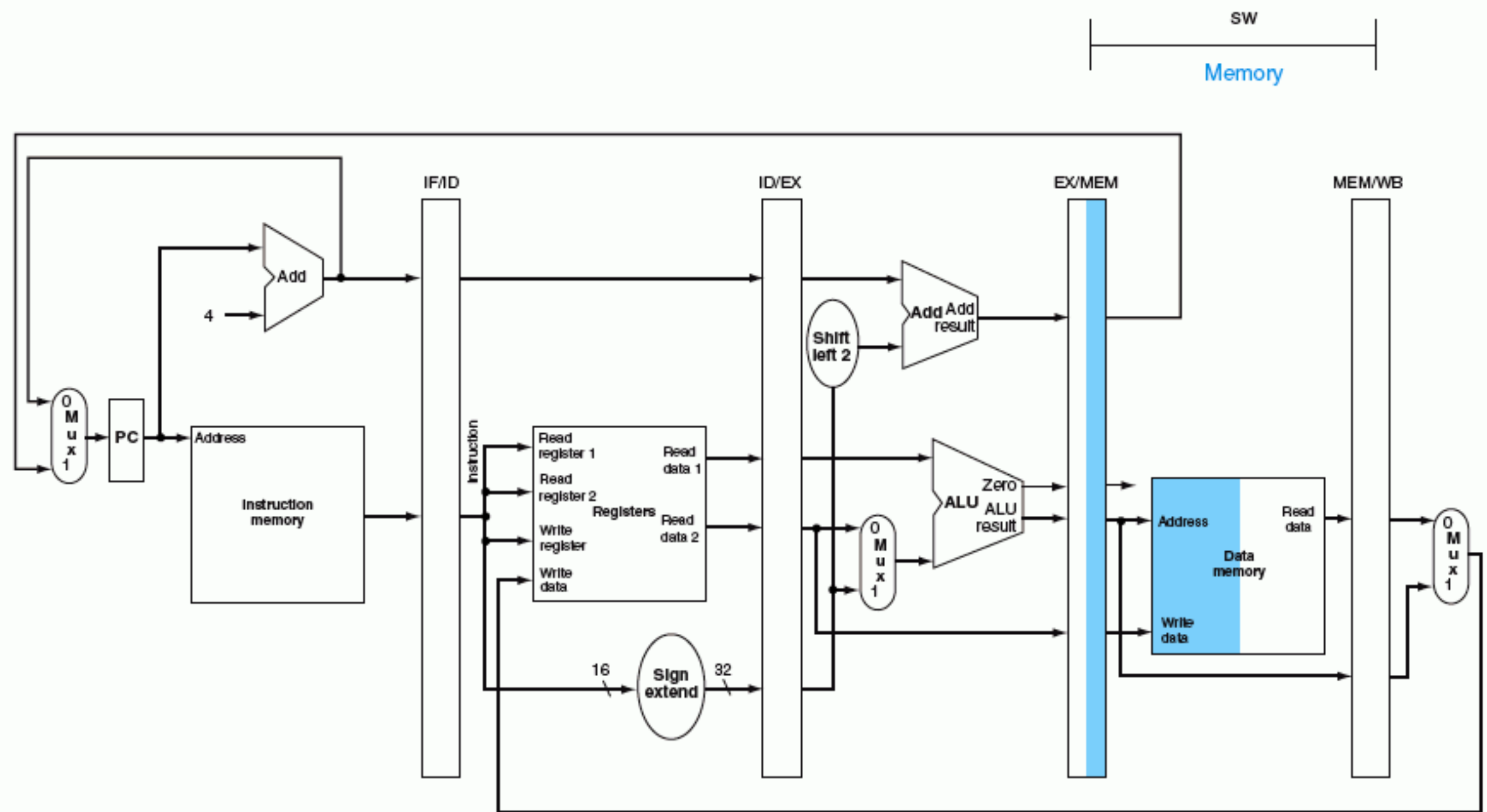


FIGURE 6.16 MEM and WB: the fourth and fifth pipe stage of a store instruction. In the fourth stage, the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

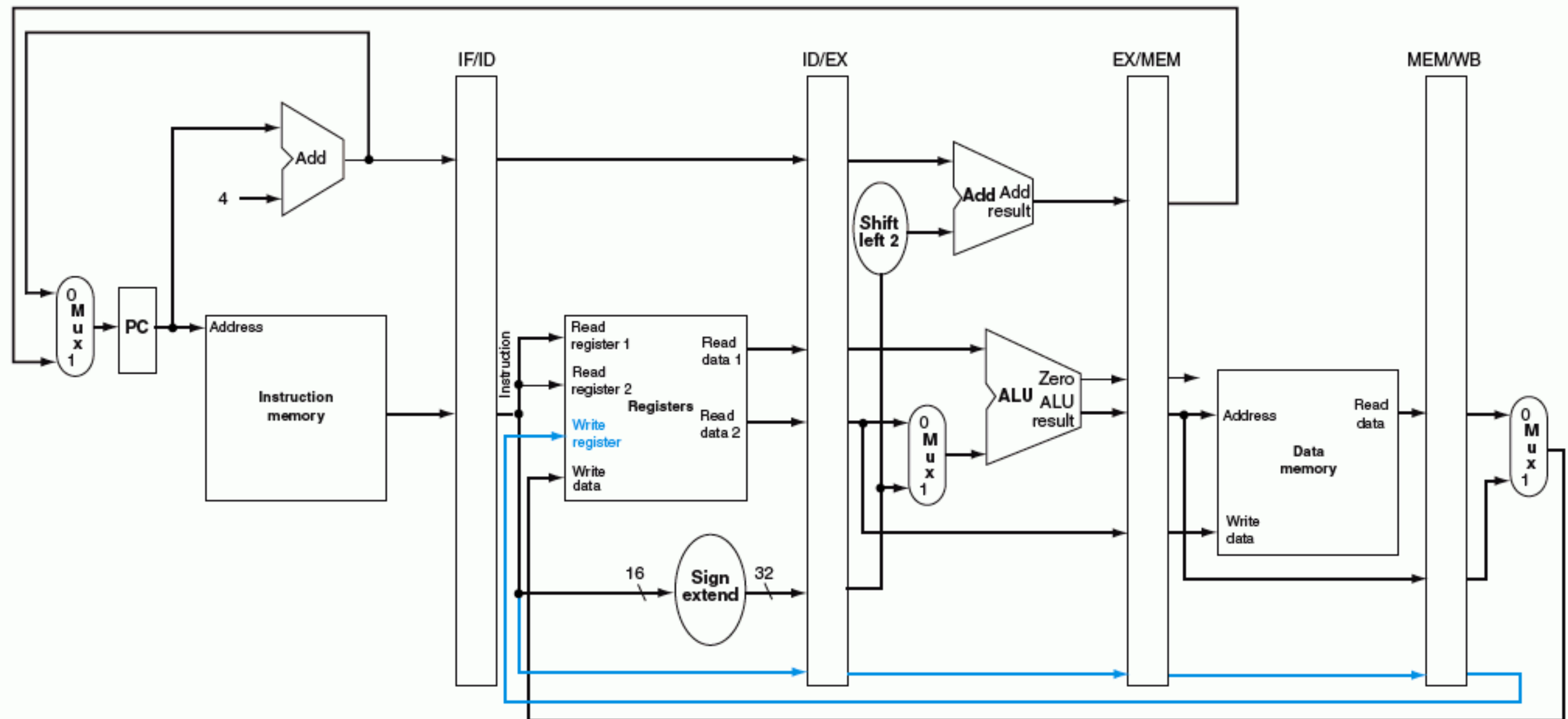


FIGURE 6.17 The corrected pipelined datapath to properly handle the load instruction. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding 5 more bits to the last three pipeline registers. This new path is shown in color.

the WB stage. Another way to think about the passing of the register number is that, in order to share the pipelined datapath, we needed to preserve the instruction read during the IF stage, so each pipeline register contains a portion of the instruction needed for that stage and later stages.

Figure 6.17 shows the correct version of the datapath, passing the write register number first to the ID/EX register, then to the EX/MEM register, and finally to the MEM/WB register. The register number is used during the WB stage to specify the register to be written. Figure 6.18 is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load word instruction in Figures 6.12 through 6.14. See Section 6.6 for an explanation of how to make the branch instruction work as expected.

Graphically Representing Pipelines

Pipelining can be difficult to understand, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as Figure 6.10 on page 387, and *single-clock-cycle pipeline diagrams*,

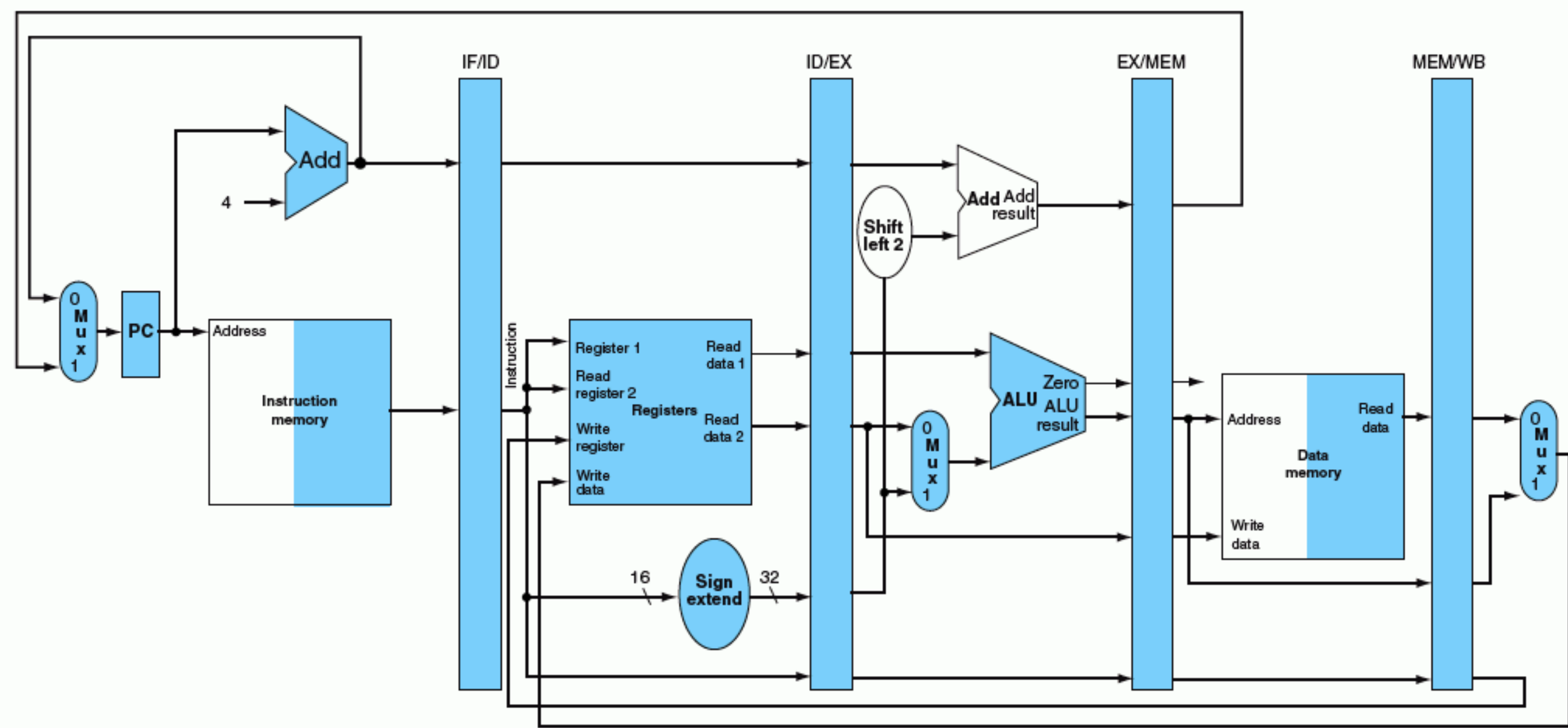


FIGURE 6.18 The portion of the datapath in Figure 6.17 that is used in all five stages of a load instruction.

such as Figures 6.12 through 6.16. The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
lw    $t0, 20($t1)
sub    $t1, $t2, $t3
add    $t2, $t3, $t4
lw    $t3, 24($t1)
add    $t4, $t5, $t6
```

Figure 6.19 shows the multiple-clock-cycle pipeline diagram for these instructions. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom of the page, similar to the laundry pipeline in Figure 6.1 on page 371. A representation of the pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles. These stylized datapaths represent the five stages of our pipeline, but a rectangle naming each pipe stage works just as well. Figure 6.20 shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that Figure 6.19 shows the physical resources used at each stage, while Figure 6.20 uses the *name* of each stage. We use multiple-clock-cycle diagrams to give overviews of pipelining situations.

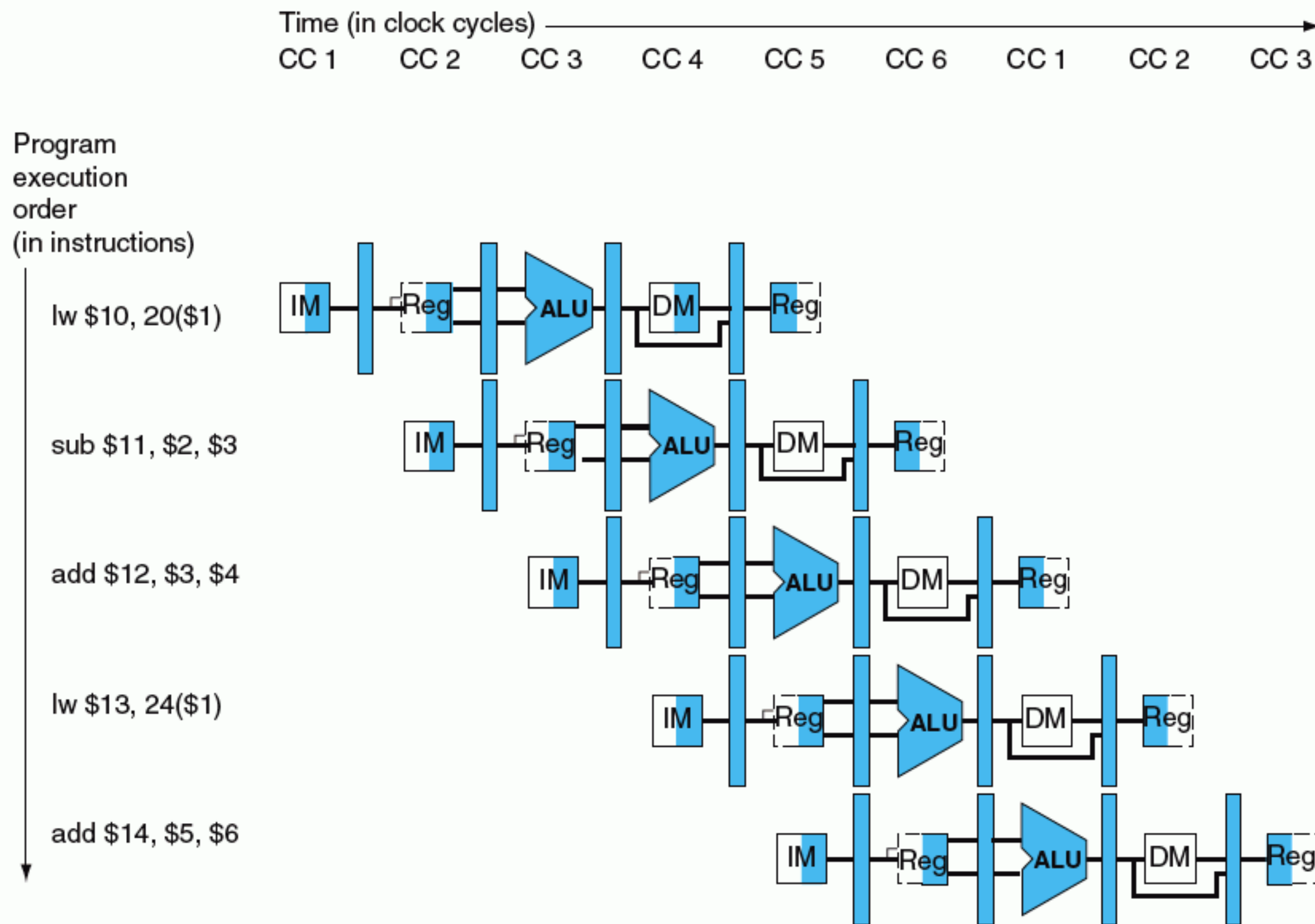


FIGURE 6.19 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 6.4, here we show the pipeline registers between each stage. Figure 6.20 shows the traditional way to draw this diagram.

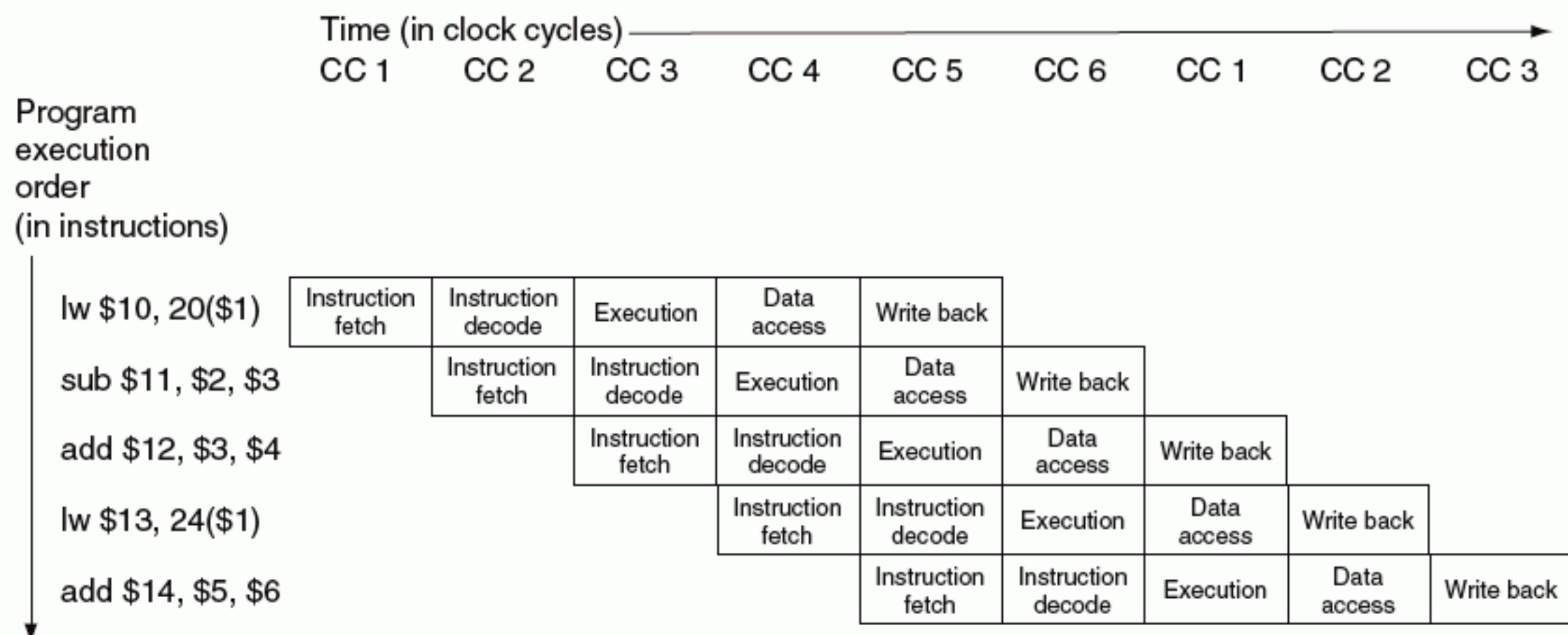


FIGURE 6.20 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 6.19.

Single-clock-cycle pipeline diagrams show the state of the entire datapath during a single clock cycle, and usually all five instructions in the pipeline are identified by labels above their respective pipeline stages. We use this type of figure to show the details of what is happening within the pipeline during each clock cycle; typically, the drawings appear in groups to show pipeline operation over a sequence of clock cycles. A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagram, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle. For example, Figure 6.21 shows the single-clock-cycle diagram corresponding to clock cycle 5 of Figures 6.19 and 6.20. Obviously, the single-clock-cycle diagrams have more detail and take significantly more space to show the same number of clock cycles. The For More Practice section included on the CD includes the corresponding single-clock-cycle diagrams for these two instructions as well as exercises asking you to create such diagrams for another code sequence.

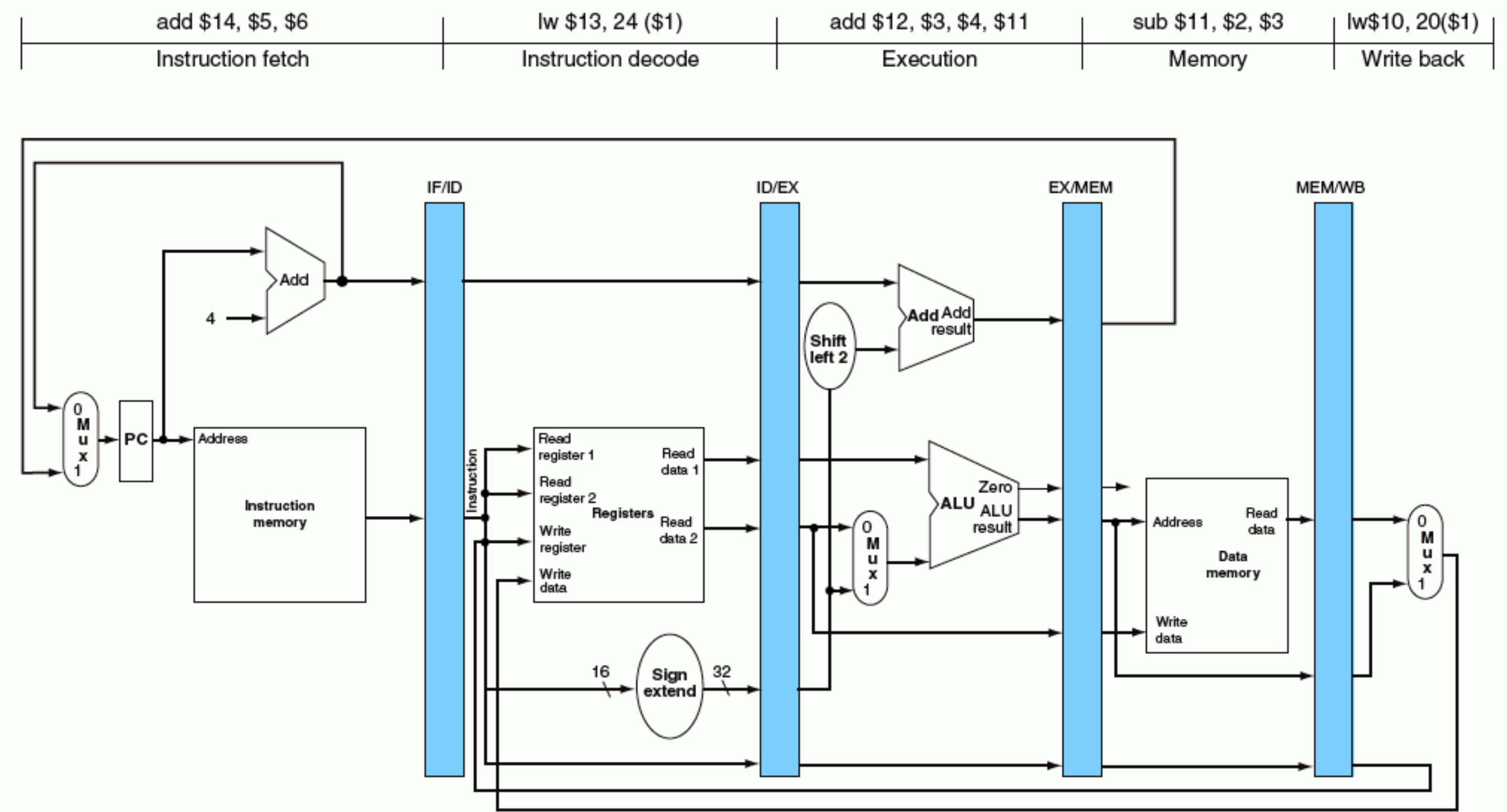


FIGURE 6.21 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 6.19 and 6.20. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

A group of students have been debating the efficiency of the five-stage pipeline when one student pointed out that not all instructions are active in every stage of the pipeline. After deciding to ignore the effects of hazards, they made the following five statements. Which ones are correct?

1. Allowing jumps, branches, and ALU instructions to take fewer stages than the five required by the load instruction will increase pipeline performance under all circumstances.
2. Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle; the number of pipe stages per instruction affects latency, not throughput.
3. Allowing jumps, branches, and ALU operations to take fewer cycles only helps when no loads or stores are in the pipeline, so the benefits are small.
4. You cannot make ALU instructions take fewer cycles because of the write-back of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.
5. Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter. This could improve performance.

Check Yourself

6.3 Pipelined Control

Just as we added control to the simple datapath in Section 5.4, we now add control to the pipelined datapath. We start with a simple design that views the problem through rose-colored glasses; in Sections 6.4 through 6.8, we remove these glasses to reveal the hazards of the real world.

The first step is to label the control lines on the existing datapath. Figure 6.22 shows those lines. We borrow as much as we can from the control for the simple datapath in Figure 5.17 on page 307. In particular, we use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines. These functions are defined in Figure 5.12 on page 302, Figure 5.16 on page 306, and Figure 5.18 on page 308. We reproduce the key information in Figures 6.23 through 6.25 to make the remaining text easier to follow.

As for the single-cycle implementation discussed in Chapter 5, we assume that the PC is written on each clock cycle, so there is no separate write signal for the

In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.

James Thornton,
*Design of a Computer:
The Control Data 6600, 1970*

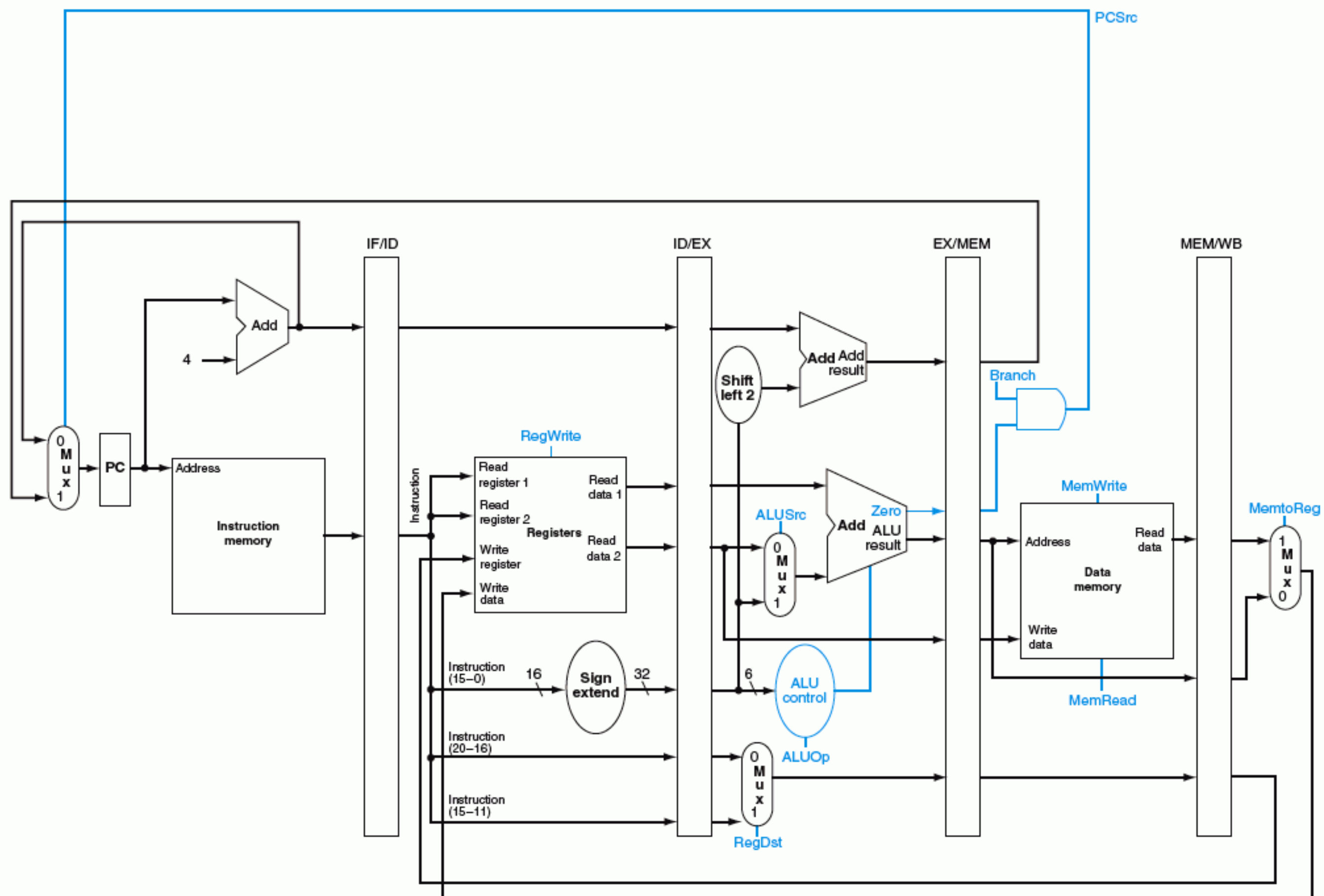


FIGURE 6.22 The pipelined datapath of Figure 6.17 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Chapter 5. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

PC. By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. *Instruction fetch:* The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 6.23 A copy of Figure 5.12 on page 302. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 6.24 A copy of Figure 5.16 on page 306. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 6.23. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 6.22. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURE 6.25 The values of the control lines are the same as in Figure 5.18 on page 308, but they have been shuffled into three groups corresponding to the last three pipeline stages.

2. *Instruction decode/register file read:* As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
3. *Execution/address calculation:* The signals to be set are RegDst, ALUOp, and ALUSrc (see Figures 6.23 and 6.24). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. These signals are set by the branch equal, load, and store instructions, respectively. Recall that PCSrc in Figure 6.24 selects the next sequential address unless control asserts Branch and the ALU result was zero.
5. *Write back:* The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values as before. Figure 6.25 has the same values as in Chapter 5, but now the nine control lines are grouped by pipeline stage.

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

Since the control lines start with the EX stage, we can create the control information during instruction decode. Figure 6.26 shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline in Figure 6.17 on page 395. Figure 6.27 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage.

What do you mean, why's it got to be built? It's a bypass. You've got to build bypasses.

Douglas Adams, *Hitchhikers Guide to the Galaxy*, 1979

6.4

Data Hazards and Forwarding

The examples in the previous section show the power of pipelined execution and how the hardware performs the task. It's now time to take off the rose-colored glasses and look at what happens with real programs. The instructions in Figures 6.19 through 6.21 were independent; none of them used the results calculated by any of the others. Yet in Section 6.1 we saw that data hazards are obstacles to pipelined execution.

Let's look at a sequence with many dependences, shown in color:

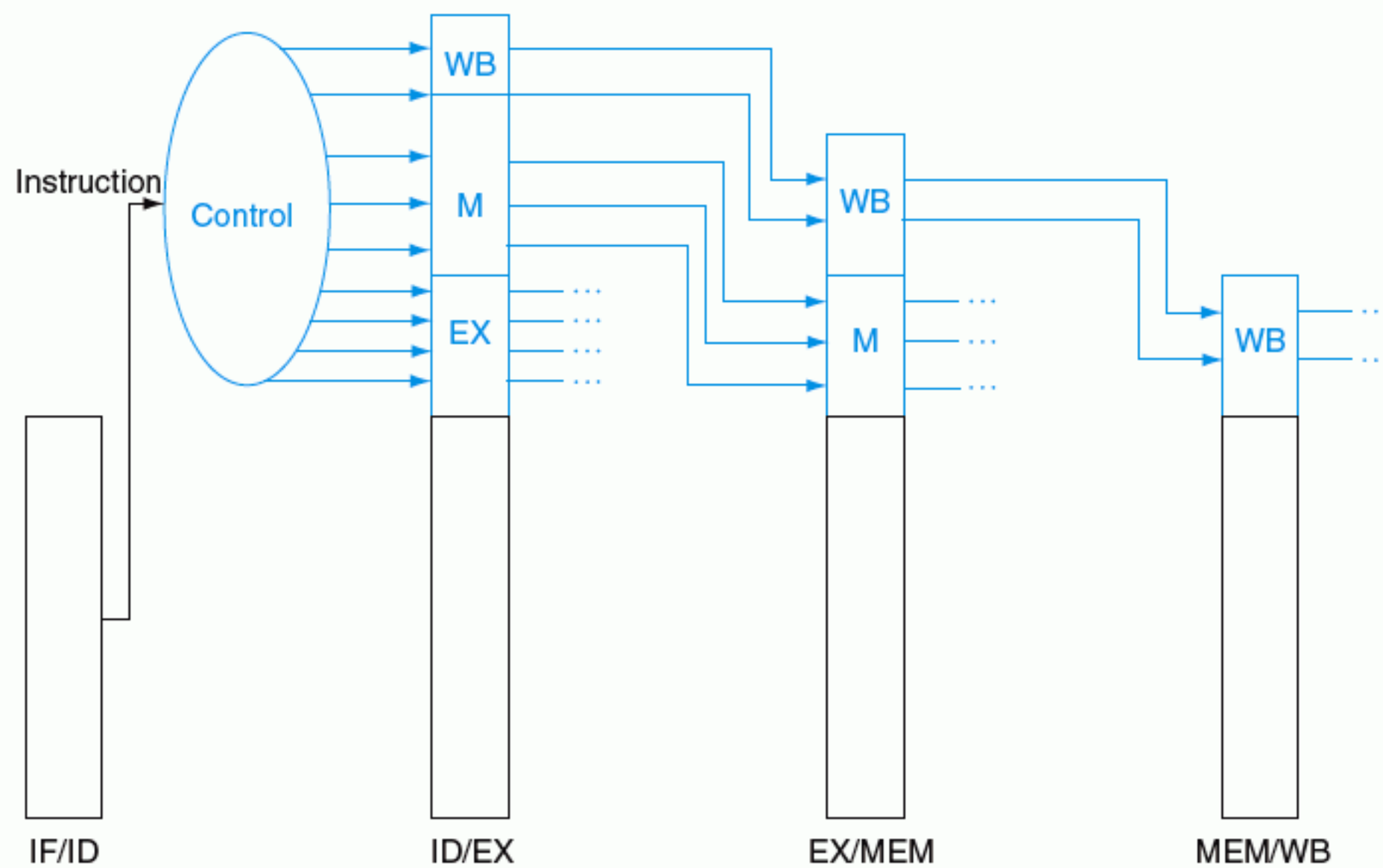


FIGURE 6.26 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

```

sub    $2, $1, $3    # Register $2 written by sub
and    $12, $2, $5    # 1st operand($2) depends on sub
or     $13, $6, $2    # 2nd operand($2) depends on sub
add    $14, $2, $2    # 1st($2) & 2nd($2) depend on sub
sw     $15, 100($2)   # Base ($2) depends on sub

```

The last four instructions are all dependent on the result in register \$2 of the first instruction. If register \$2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register \$2.

How would this sequence perform with our pipeline? Figure 6.28 illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of Figure 6.28 shows the value of register \$2, which changes during the middle of clock cycle 5, when the sub instruction writes its result.

One potential hazard can be resolved by the design of the register file hardware: what happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, we have no data hazard in this case.

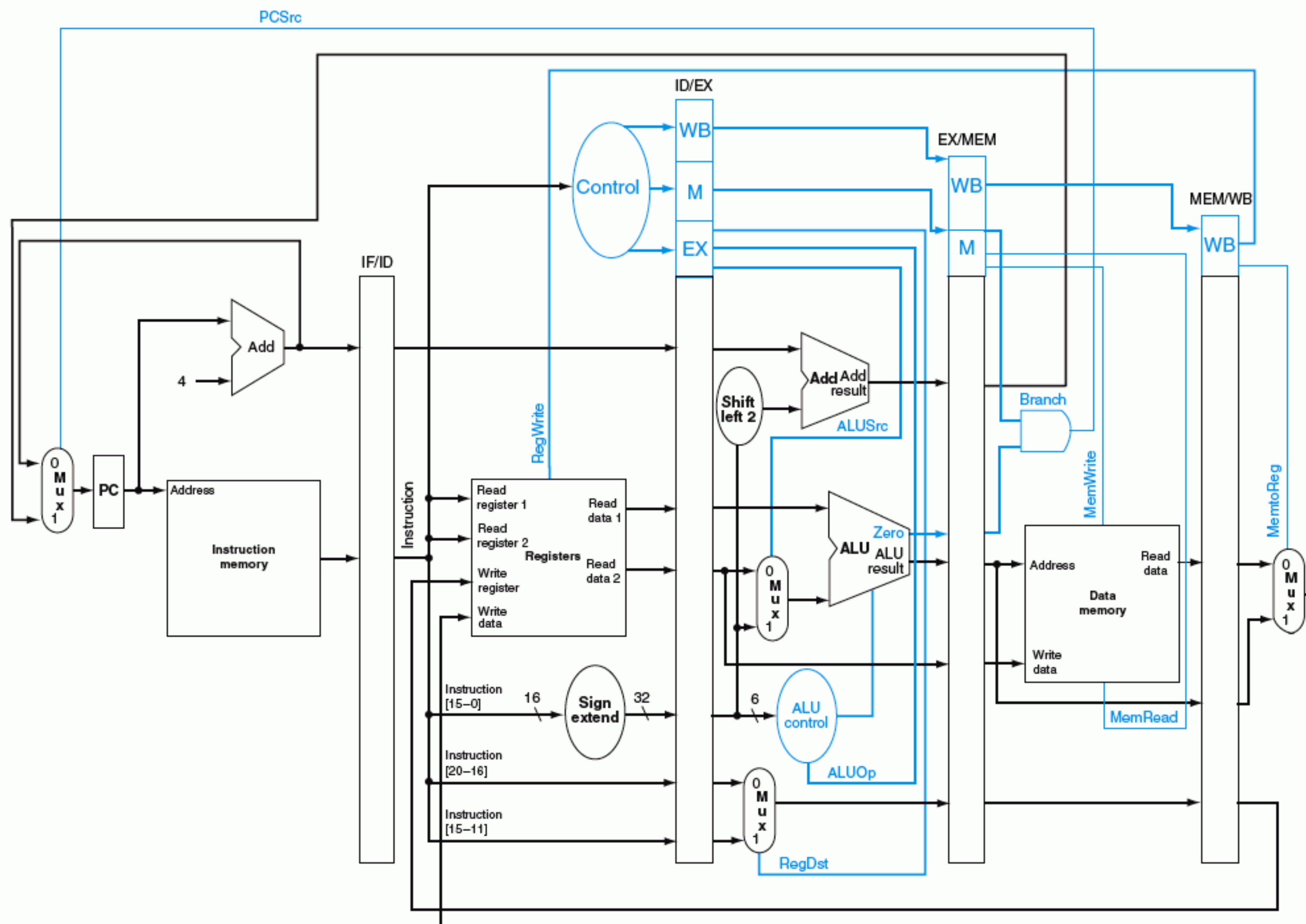


FIGURE 6.27 The pipelined datapath of Figure 6.22, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Figure 6.28 shows that the values read for register \$2 would *not* be the result of the sub instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of -20 are add and sw; the and and or instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backwards in time.

But, look carefully at Figure 6.28: When is the data from the sub instruction actually produced? The result is available at the end of the EX stage or clock cycle 3. When is the data actually needed by the and and or instructions? At the beginning of the EX stage, or clock cycles 4 and 5, respectively. Thus, we can execute

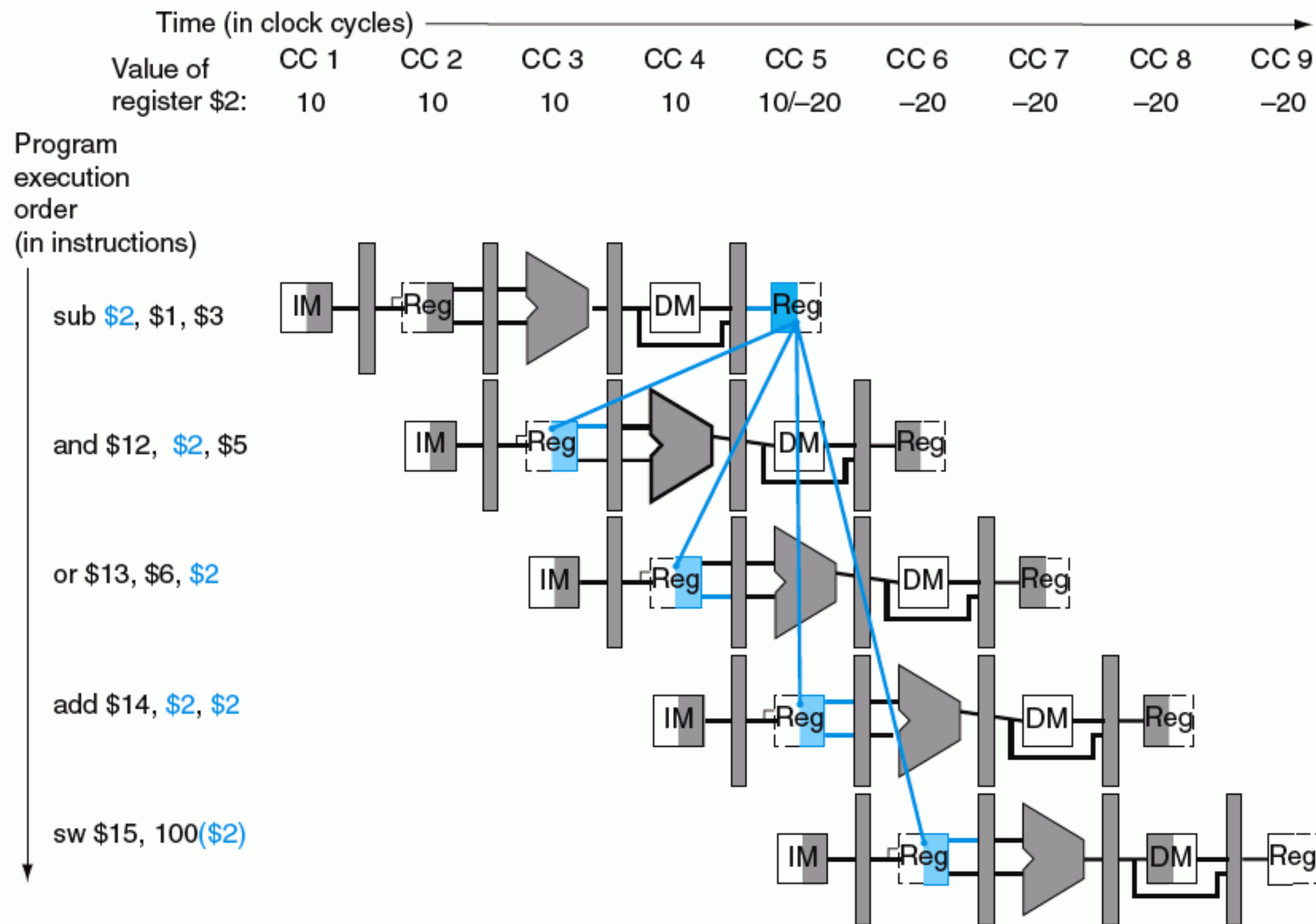


FIGURE 6.28 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and “CC i ” at the top of the figure means clock cycle i . The first instruction writes into \$2, and all the following instructions read \$2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backwards in time are *pipeline data hazards*.

this segment without stalls if we simply *forward* the data as soon as it is available to any units that need it before it is available to read from the register file.

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.

A notation that names the fields of the pipeline registers allows for a more precise notation of dependences. For example, “ID/EX.RegisterRs” refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name

of the field in that register. Using this notation, the two pairs of hazard conditions are

1a. $EX/MEM.RegisterRd = ID/EX.RegisterRs$

1b. $EX/MEM.RegisterRd = ID/EX.RegisterRt$

2a. $MEM/WB.RegisterRd = ID/EX.RegisterRs$

2b. $MEM/WB.RegisterRd = ID/EX.RegisterRt$

The first hazard in the sequence on page 403 is on register \$2, between the result of `sub $2,$1,$3` and the first read operand of `and $12,$2,$5`. This hazard can be detected when the `and` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

$$EX/MEM.RegisterRd = ID/EX.RegisterRs = \$2$$

EXAMPLE

Dependence Detection

Classify the dependences in this sequence from page 403:

```
sub    $2,    $1, $3 # Register $2 set by sub
and    $12,   $2, $5 # 1st operand($2) set by sub
or     $13,   $6, $2 # 2nd operand($2) set by sub
add    $14,   $2, $2 # 1st($2) & 2nd($2) set by sub
sw     $15,   100($2) # Index($2) set by sub
```

ANSWER

As mentioned above, the `sub-and` is a type 1a hazard. The remaining hazards are

- The `sub-or` is a type 2b hazard:
 $MEM/WB.RegisterRd = ID/EX.RegisterRt = \2
- The two dependences on `sub-add` are not hazards because the register file supplies the proper data during the ID stage of `add`.
- There is no data hazard between `sub` and `sw` because `sw` reads \$2 the clock cycle *after* `sub` writes \$2.

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it was unnecessary. One solution is simply to check to see if the `RegWrite` signal will be active: examining the WB control field

of the pipeline register during the EX and MEM stages determines if RegWrite is asserted. Also, MIPS requires that every use of \$0 as an operand must yield an operand value of zero. In the event that an instruction in the pipeline has \$0 as its destination (for example, `sll $0, $1, 2`), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for \$0 frees the assembly programmer and the compiler of any requirement to avoid using \$0 as a destination. The conditions above thus work properly as long we add `EX/MEM.RegisterRd \neq 0` to the first hazard condition and `MEM/WB.RegisterRd \neq 0` to the second.

Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

Figure 6.29 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in Figure 6.28. The change is that the dependence begins from a *pipeline* register rather than waiting for the WB stage to write the register file. Thus the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

If we can take the inputs to the ALU from *any* pipeline register rather than just ID/EX, then we can forward the proper data. By adding multiplexors to the input of the ALU and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences.

For now, we will assume the only instructions we need to forward are the four R-format instructions: `add`, `sub`, `and`, and `or`. Figure 6.30 shows a close-up of the ALU and pipeline register before and after adding forwarding. Figure 6.31 shows the values of the control lines for the ALU multiplexors that select either the register file values or one of the forwarded values.

This forwarding control will be in the EX stage because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. We already have the `rt` field (bits 20–16). Before forwarding, the ID/EX register had no need to include space to hold the `rs` field. Hence, `rs` (bits 25–21) is added to ID/EX.

Let's now write both the conditions for detecting hazards and the control signals to resolve them:

1. EX hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd  $\neq$  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd  $\neq$  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

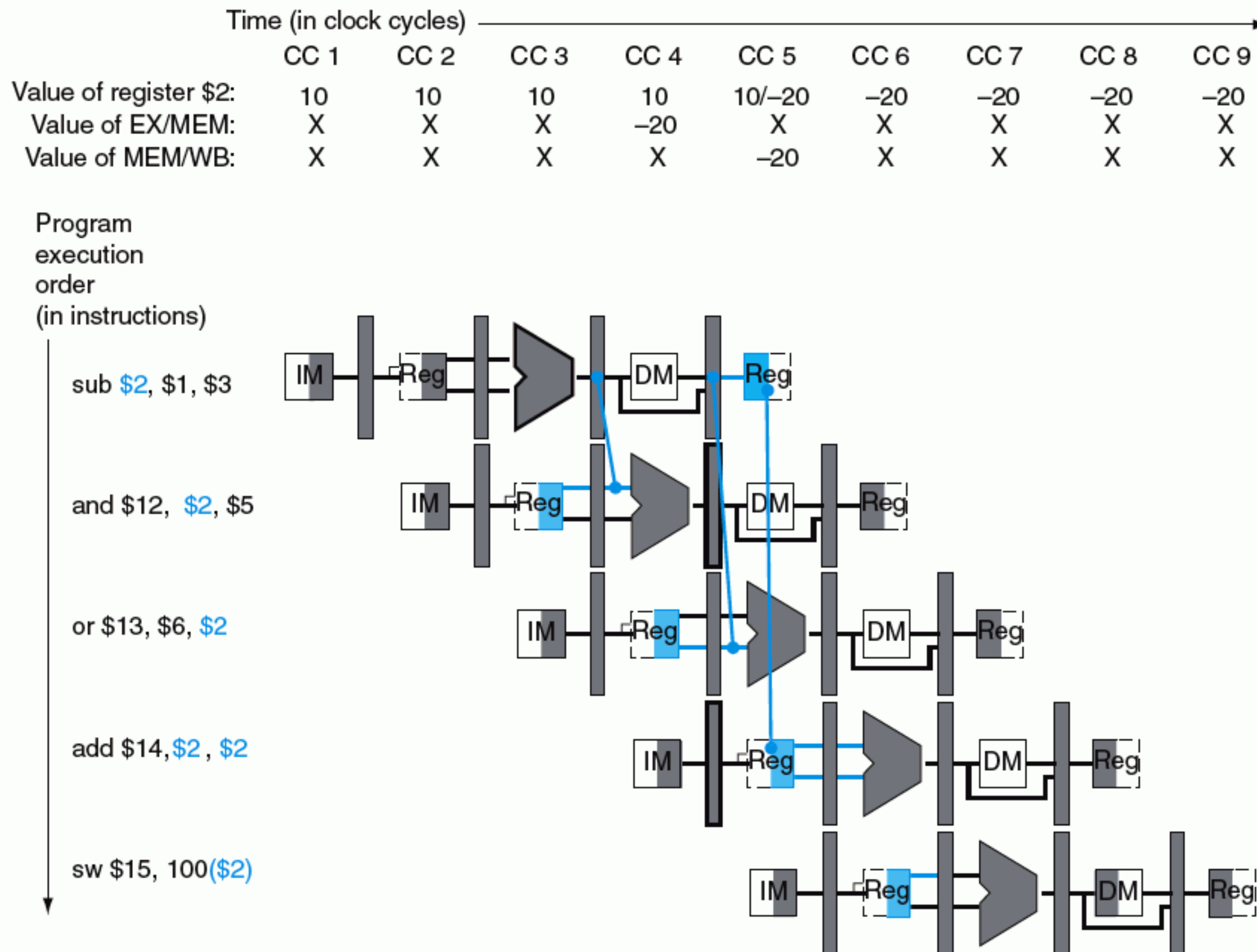
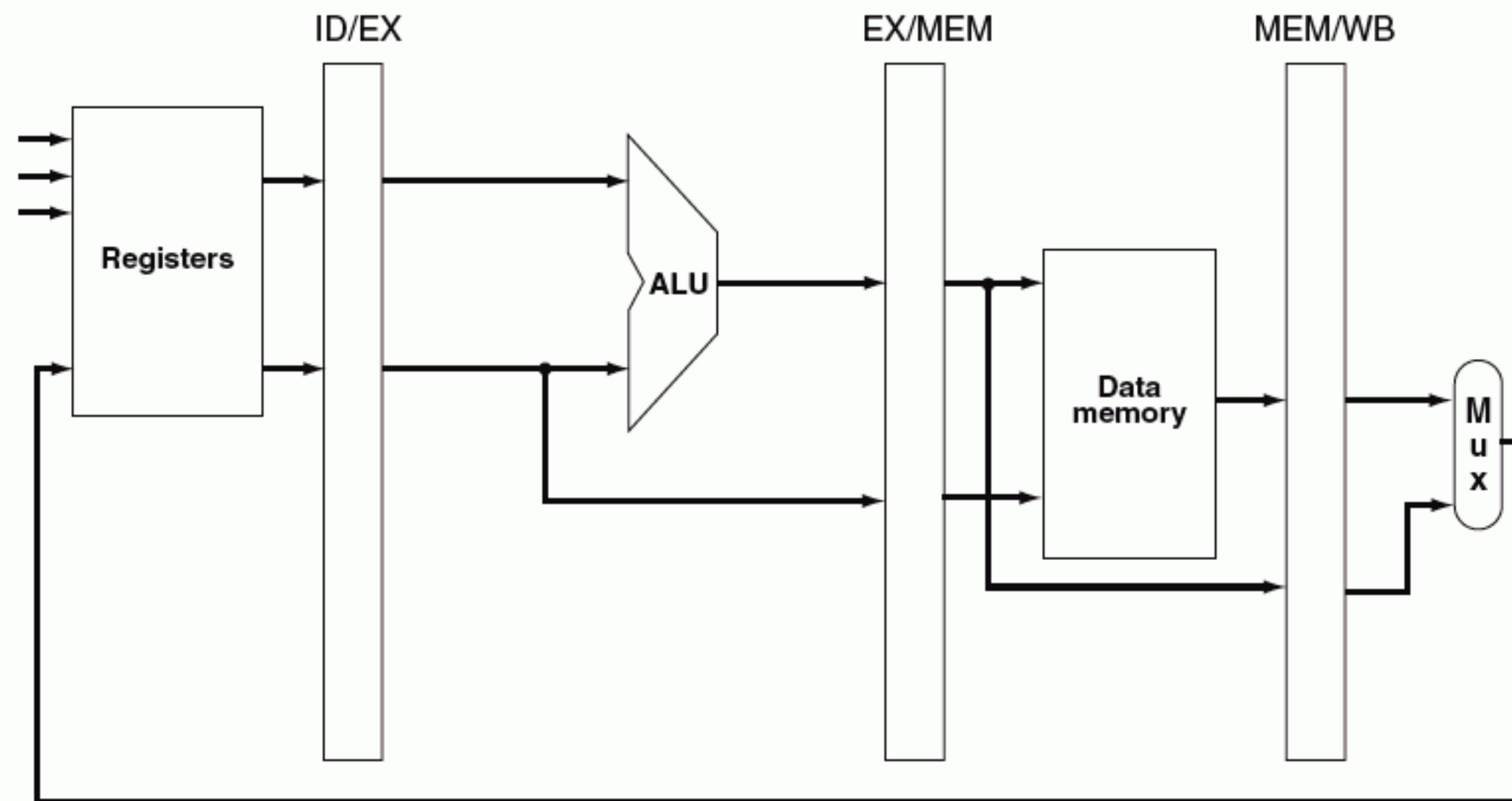
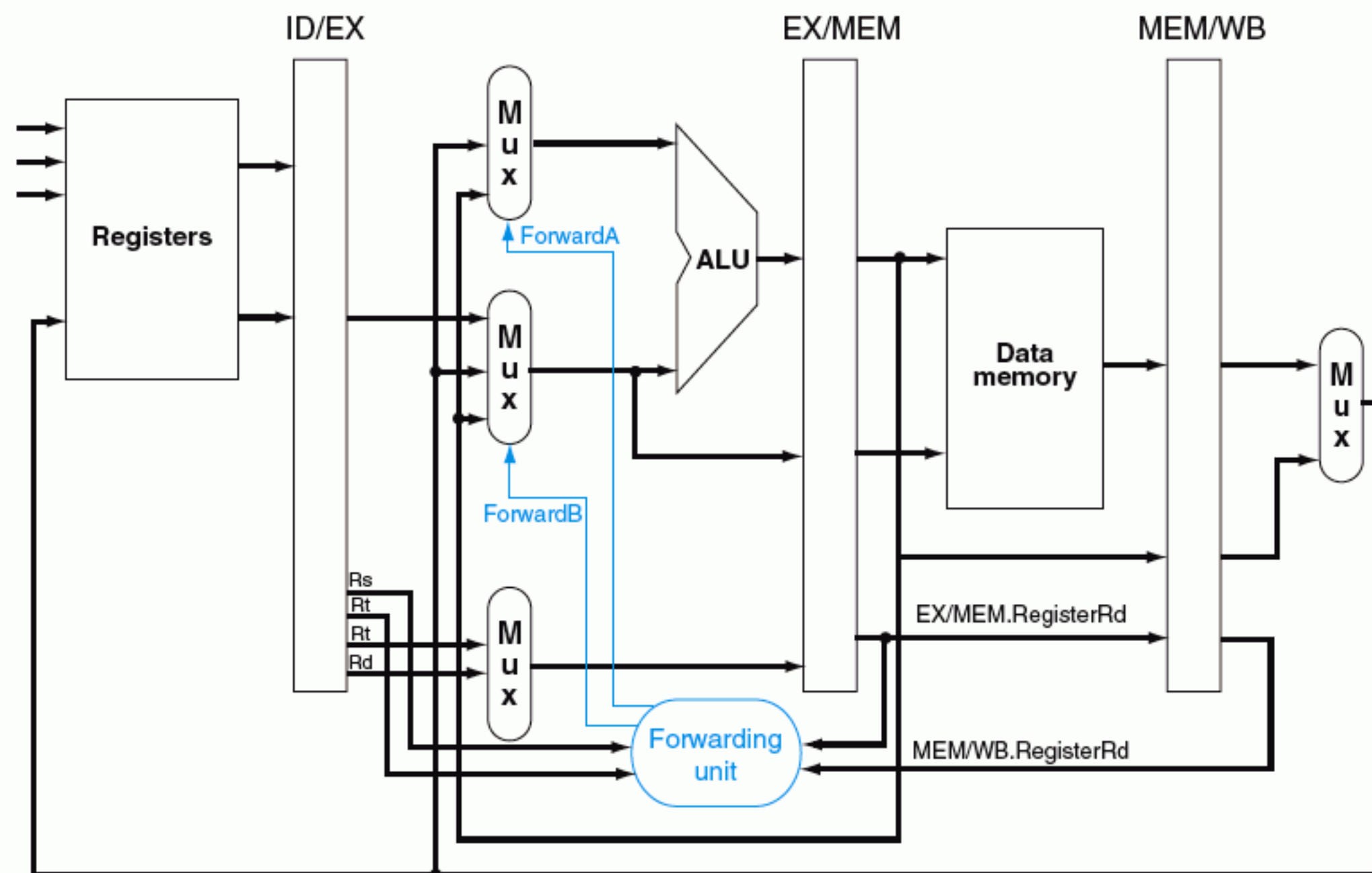



FIGURE 6.29 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the `and` instruction and `or` instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the `add` does not stall, but the values come from the register file instead of a pipeline register. Register file “forwarding”—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register \$2 having the value 10 at the beginning and -20 at the end of the clock cycle. As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.



a. No forwarding



b. With forwarding

FIGURE 6.30 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware. Note that the ID/EX.RegisterRt field is shown twice, once to connect to the mux and once to the forwarding unit, but it is a single signal. As in the earlier discussion, this ignores forwarding of a store value to a store instruction.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 6.31 The control values for the forwarding multiplexors in Figure 6.30. The signed immediate that is another input to the ALU is described in the elaboration at the end of this section.

2. MEM hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

As mentioned above, there is no hazard in the WB stage because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage. For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .
```

In this case, the result is forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus the control for the MEM hazard would be (with the additions highlighted)

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
```

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Figure 6.32 shows the hardware necessary to support forwarding for operations that use results during the EX stage.

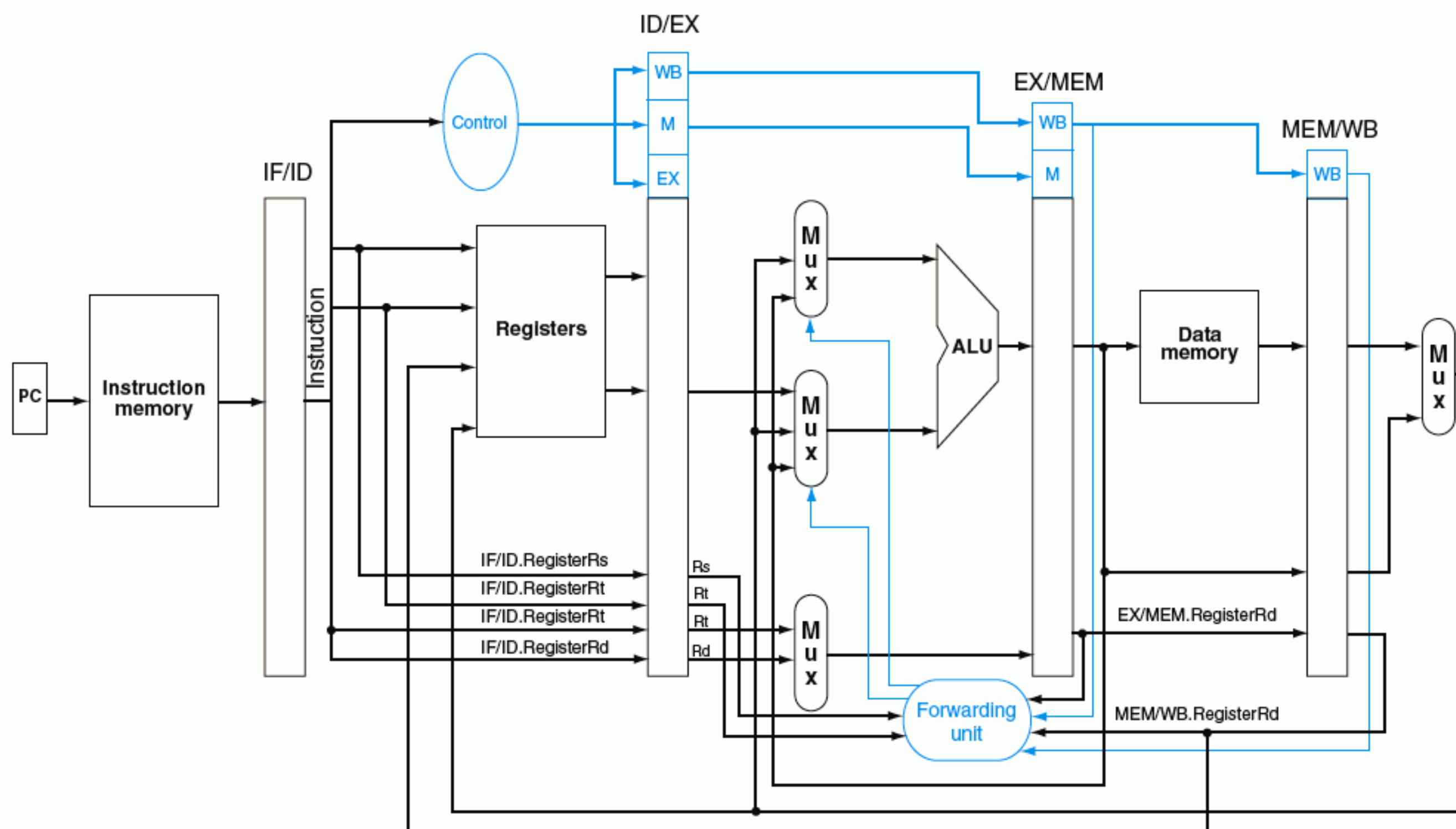


FIGURE 6.32 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 6.27 on page 404, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath such as the branch hardware and the sign extension hardware.

Elaboration: Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. But consider loads immediately followed by stores. We need to add more forwarding hardware to make memory-to-memory copies run faster. If we were to redraw Figure 6.29 on page 408, replacing the `sub` and `and` instructions by `lw` and an `sw`, we would see that it is possible to avoid a stall, since the data exists in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option. We leave this modification as an exercise.

In addition, the signed-immediate input to the ALU, needed by loads and stores, is missing from the datapath in Figure 6.32 on page 411. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register input to the ALU, the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate. Figure 6.33 shows this addition. Note that this solution differs from what we learned in Chapter 5, where the multiplexor controlled by line `ALUSrcB` was expanded to include the immediate input.

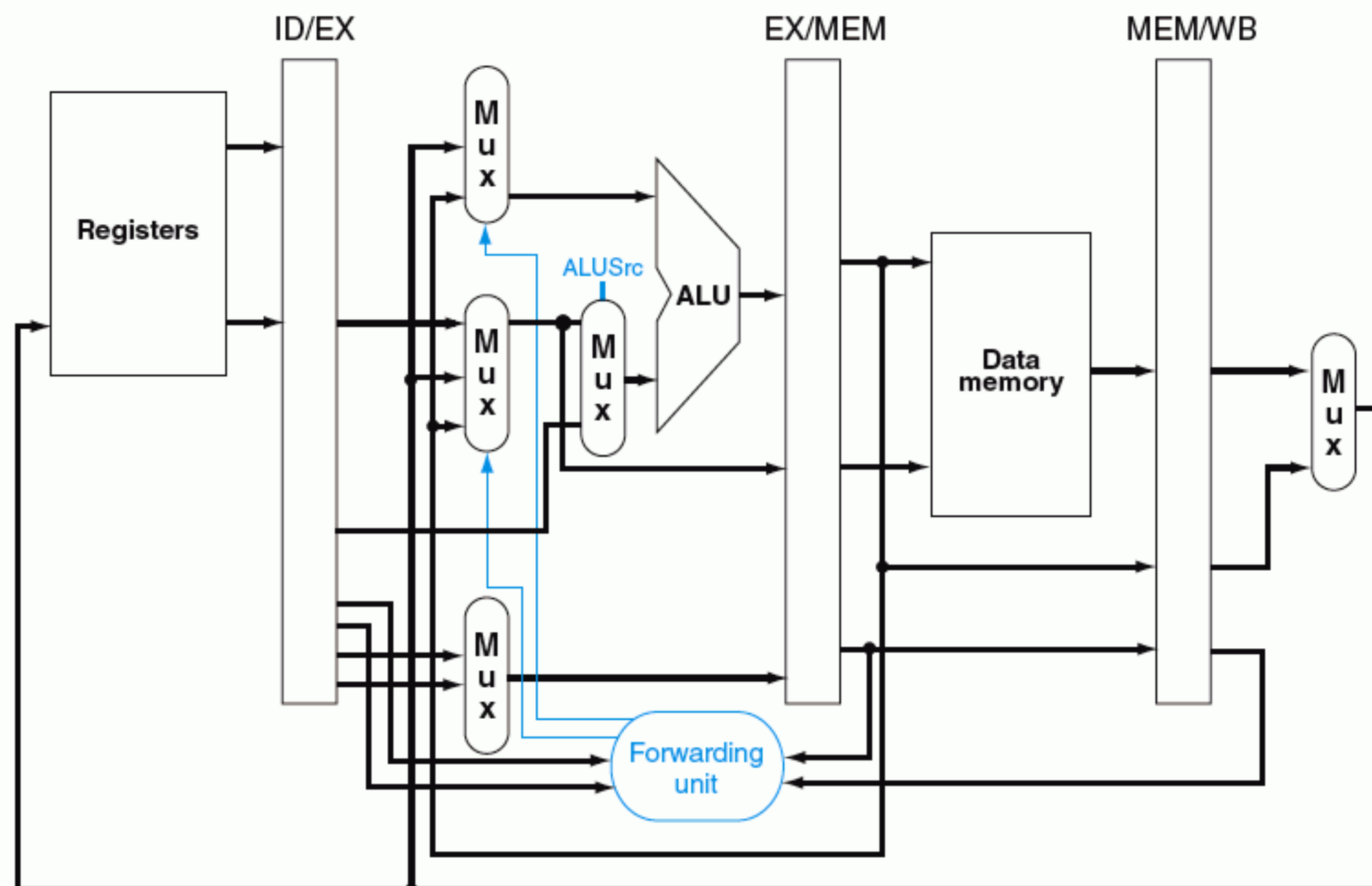


FIGURE 6.33 A close-up of the datapath in Figure 6.30 on page 409 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

6.5

Data Hazards and Stalls

*If at first you don't succeed,
redefine success.*

Anonymous

As we said in Section 6.1, one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. Figure 6.34 illustrates the problem. The data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

Hence, in addition to a forwarding unit, we need a *hazard detection unit*. It operates during the ID stage so that it can insert the stall between the load and its use. Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage. If the condition holds, the instruction stalls 1 clock cycle. After this 1-cycle stall, the forwarding logic can handle the dependence and execution proceeds. (If there were no forwarding, then the instructions in Figure 6.34 would need another stall cycle.)

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register. Returning to our favorite analogy, it's as if you restart the washer with the same clothes and let the dryer continue tumbling empty. Of course, like the dryer, the back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: **nops**.

How can we insert these nops, which act like bubbles, into the pipeline? In Figure 6.25 on page 401, we see that deasserting all nine control signals (setting them to 0) in the EX, MEM, and WB stages will create a “do nothing” or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into

nop An instruction that does no operation to change state.

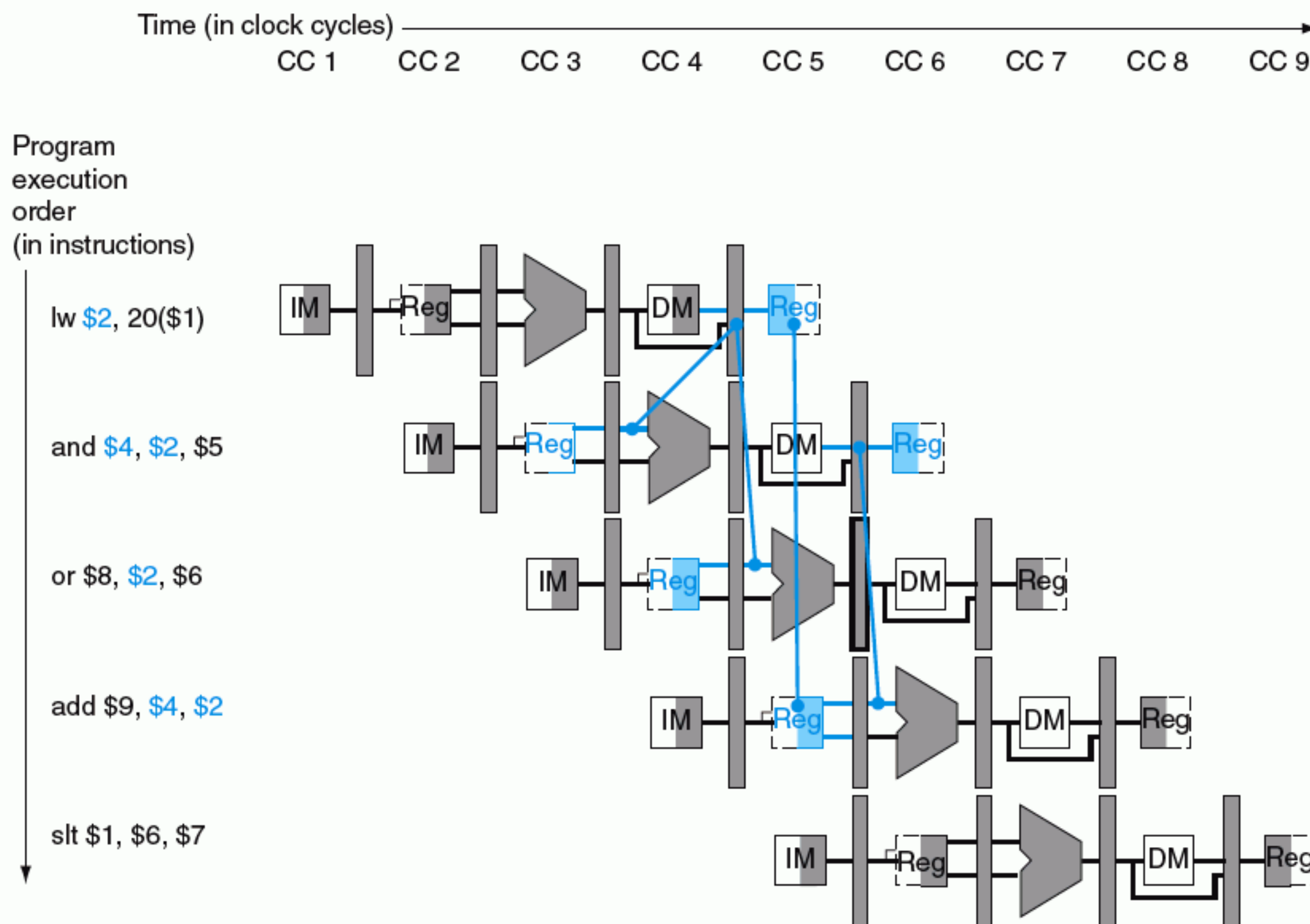


FIGURE 6.34 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (*and*) goes backwards in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These benign control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.

Figure 6.35 shows what really happens in the hardware: the pipeline execution slot associated with the *and* instruction is turned into a *nop* and all instructions beginning with the *and* instruction are delayed one cycle. The hazard forces the *and* and *or* instructions to repeat in clock cycle 4 what they did in clock cycle 3: *and* reads registers and decodes, and *or* is refetched from instruction memory. Such repeated work is what a stall looks like, but its effect is to stretch the time of the *and* and *or* instructions and delay the fetch of the *add* instruction. Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each cycle until it exits at the end.

Figure 6.36 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value

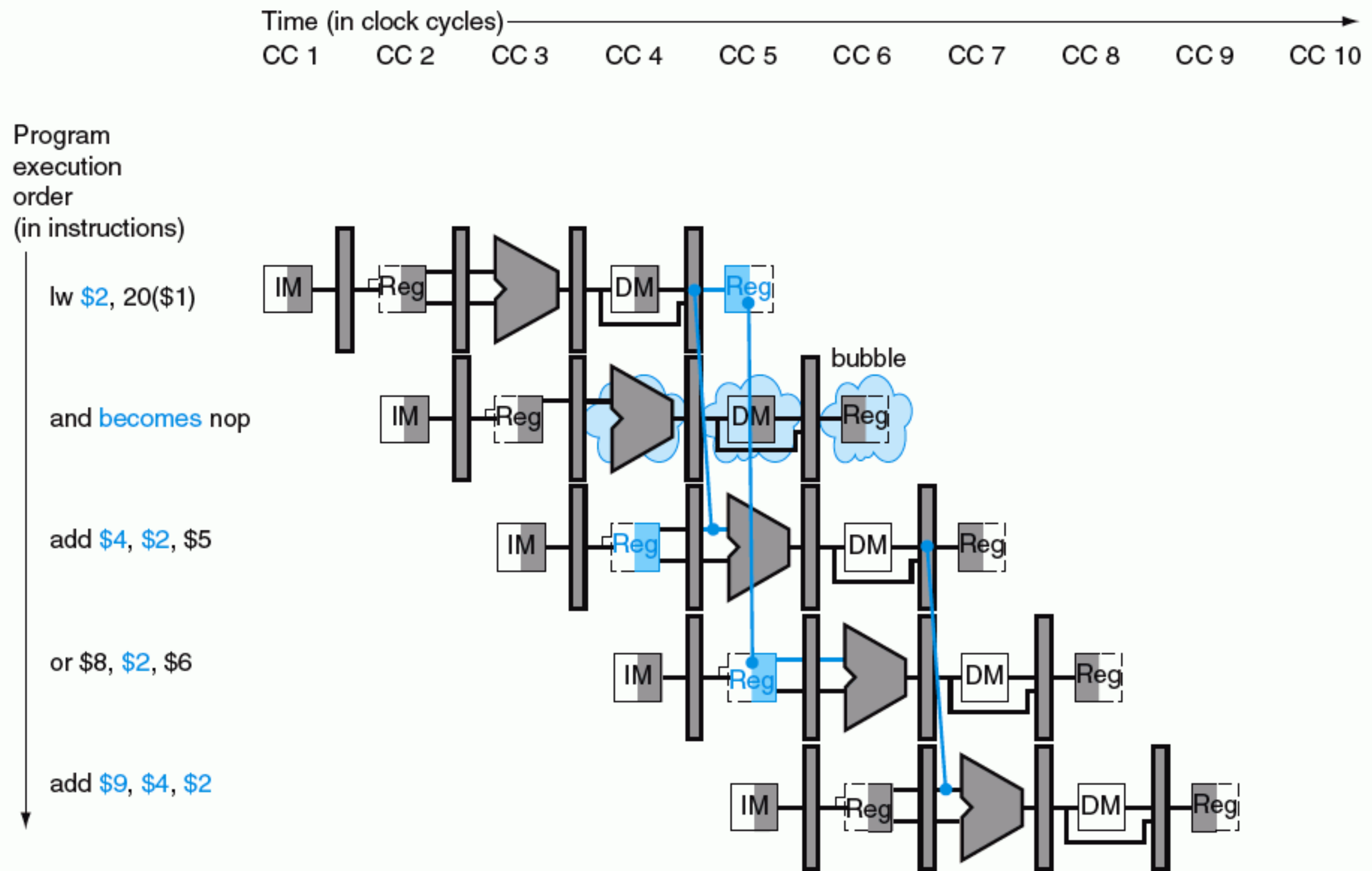


FIGURE 6.35 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a `nop`. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the `or` instruction is fetched in clock cycle 3, but its IF stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true. We show the single-clock-cycle diagrams in the [For More Practice](#) section on the CD.

Although the hardware may or may not rely on the compiler to resolve hazard dependences to ensure correct execution, the compiler must understand the pipeline to achieve the best performance. Otherwise, unexpected stalls will reduce the performance of the compiled code.

**The BIG
Picture**

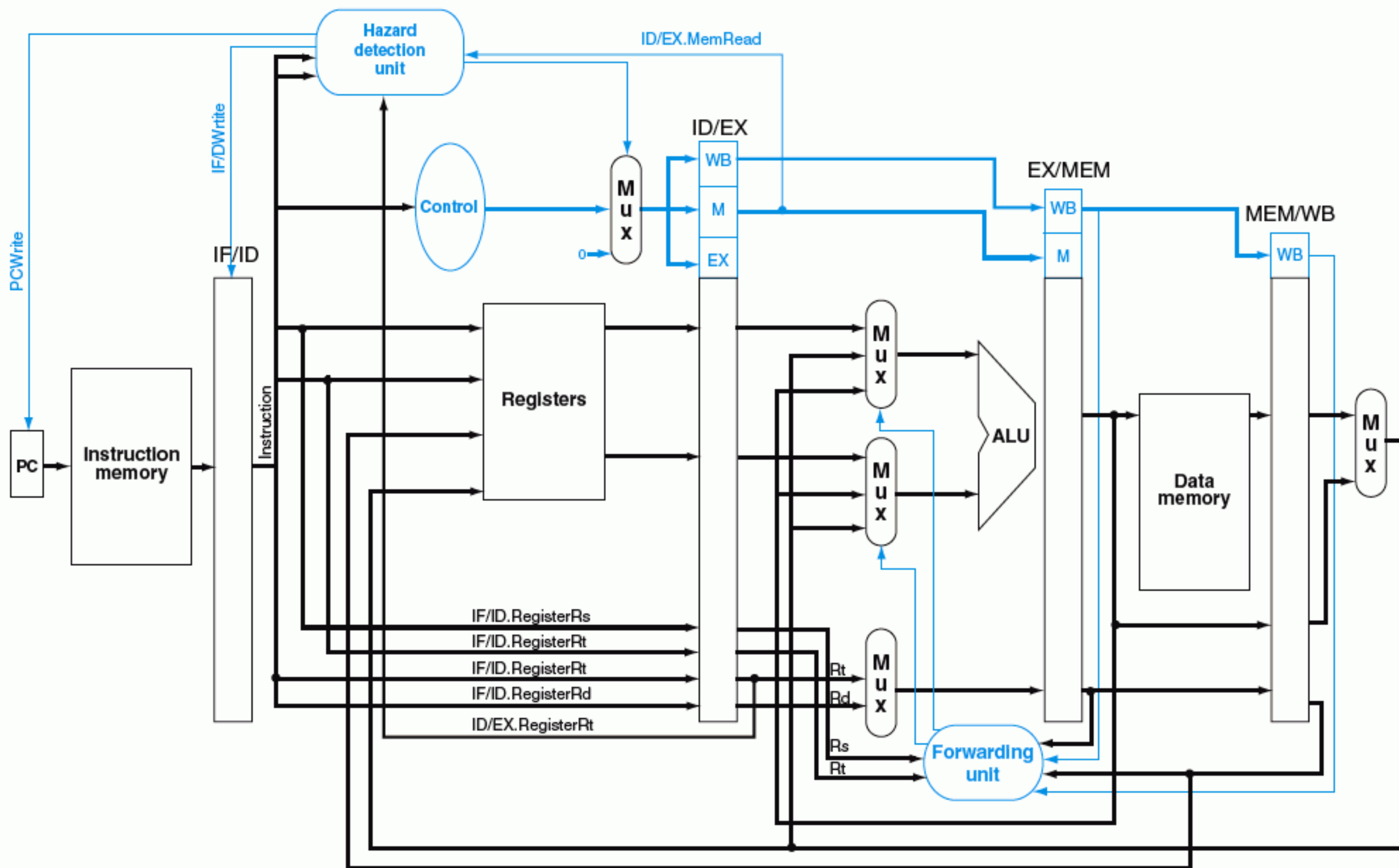


FIGURE 6.36 Pipelined control overview, showing the two multiplexers for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Elaboration: Regarding the remark earlier about setting control lines to 0 to avoid writing registers or memory: only the signals RegWrite and MemWrite need be 0, while the other control signals can be don't cares.

*There are a thousand hack-
ing at the branches of evil to
one who is striking at
the root.*

Henry David Thoreau,
Walden, 1854

6.6

Branch Hazards

Thus far we have limited our concern to hazards involving arithmetic operations and data transfers. But as we saw in Section 6.1, there are also pipeline hazards involving branches. Figure 6.37 shows a sequence of instructions and indicates

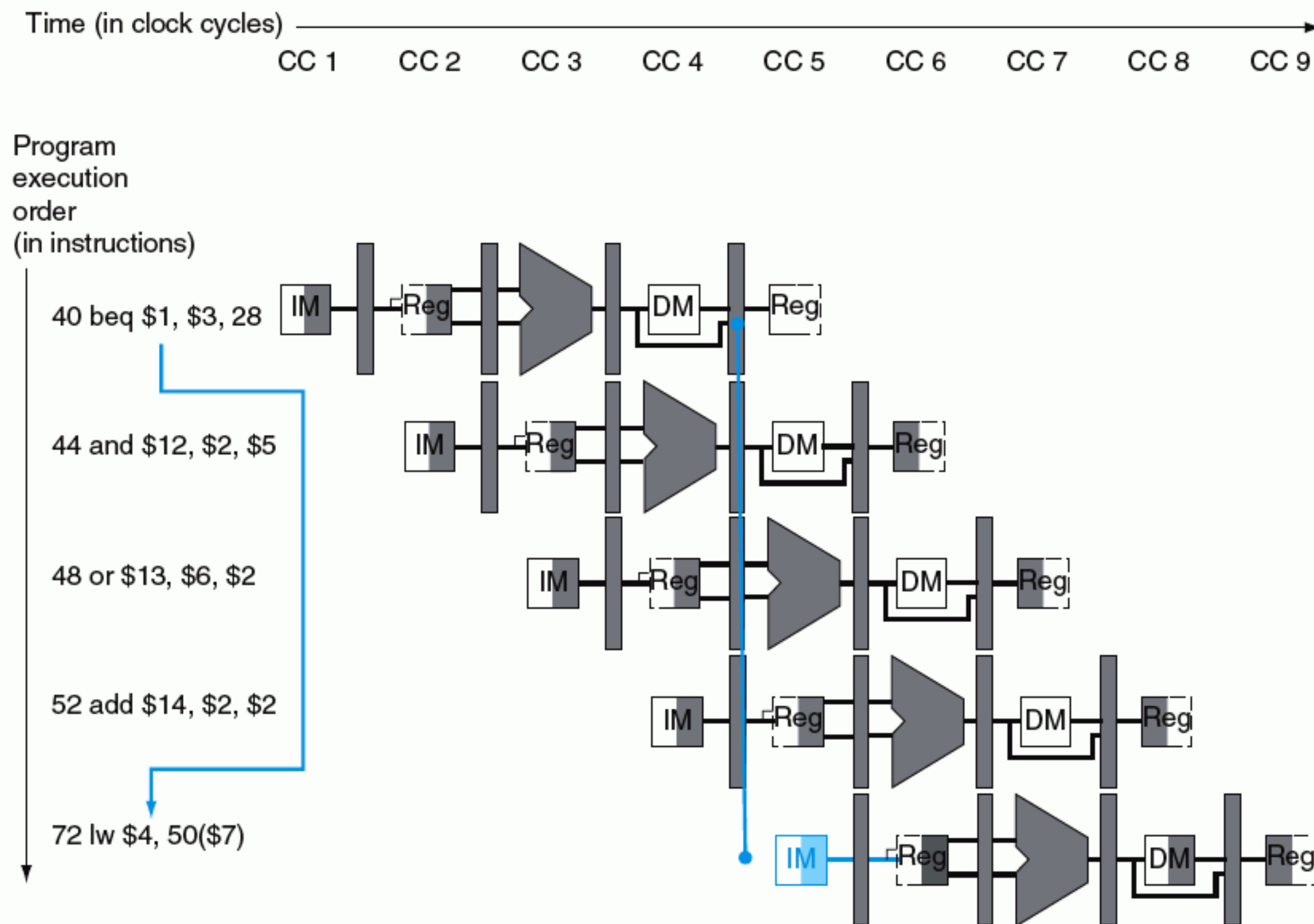


FIGURE 6.37 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72. (Figure 6.7 on page 380 assumed extra hardware to reduce the control hazard to 1 clock cycle; this figure uses the nonoptimized datapath.)

when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. As mentioned in Section 6.1, this delay in determining the proper instruction to fetch is called a *control hazard* or *branch hazard*, in contrast to the *data hazards* we have just examined.

This section on control hazards is shorter than the previous sections on data hazards. The reasons are that control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is for data hazards. Hence, we use simpler schemes. We look at two schemes for resolving control hazards and one optimization to improve these schemes.

Assume Branch Not Taken

As we saw in Section 6.1, stalling until the branch is complete is too slow. A common improvement over branch stalling is to assume that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, we merely change the original control values to 0s, much as we did to stall for a load-use data hazard. The difference is that we must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage; for load-use stalls, we just changed control to 0 in the ID stage and let them percolate through the pipeline. Discarding instructions, then, means we must be able to **flush instructions** in the IF, ID, and EX stages of the pipeline.

flush (instructions) To discard instructions in a pipeline, usually due to an unexpected event.

Reducing the Delay of Branches

One way to improve branch performance is to reduce the cost of the taken branch. Thus far we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty. The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates. When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required—a situation that is similar to the use of condition codes for branches.

Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization. For example, to implement branch-on-equal (and its inverse), we will need to forward results to the equality test logic that operates during ID. There are two complicating factors:

1. During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.
2. Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch.

Despite these difficulties, moving the branch execution to the ID stage is an improvement since it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched. The exercises explore the details of implementing the forwarding path and detecting the hazard.

To flush instructions in the IF stage, we add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a nop, an instruction that has no action and changes no state.

Pipelined Branch

Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken and that we moved the branch execution to the ID stage:

```

36 sub $t0, $t4, $t8
40 beq  $t1, $t3, 7 # PC-relative branch to 40 + 4
   + 7 * 4 = 72
44 and $t2, $t2, $t5
48 or  $t3, $t2, $t6
52 add $t4, $t4, $t2
56 slt $t5, $t6, $t7
   . . .
72 lw  $t4, 50($t7)

```

Figure 6.38 shows what happens when a branch is taken. Unlike Figure 6.37, there is only one pipeline bubble on a taken branch.

EXAMPLE

ANSWER

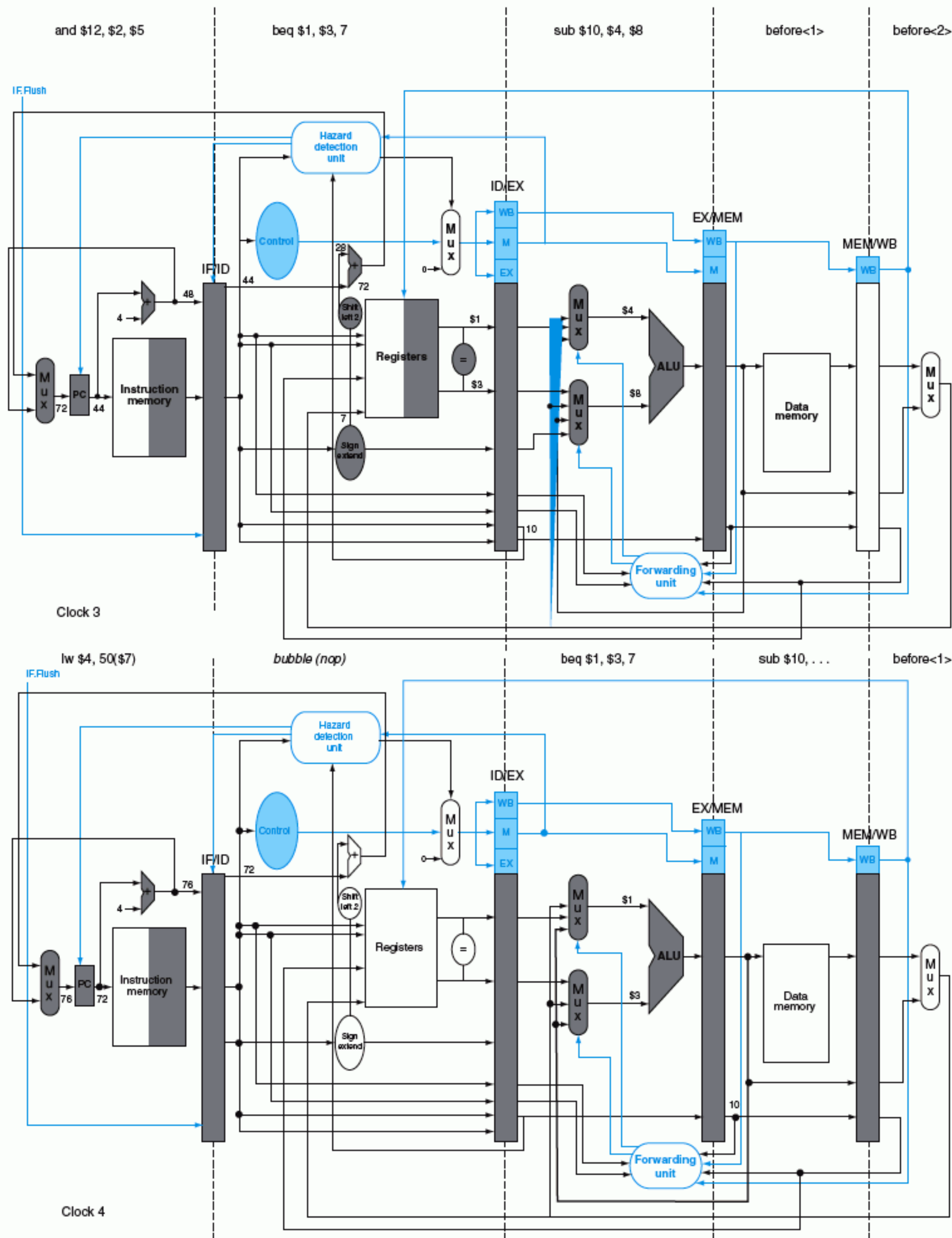


FIGURE 6.38 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or nop instruction in the pipeline as a result of the taken branch. (Since the nop is really `slil $0, $0, 0`, it's arguable whether or not the ID stage in clock 4 should be highlighted.)

Dynamic Branch Prediction

Assuming a branch is not taken is one simple form of *branch prediction*. In that case, we predict that branches are untaken, flushing the pipeline when we are wrong. For the simple five-stage pipeline, such an approach, possibly coupled with compiler-based prediction, is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issue, the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance. As we mentioned in Section 6.1, with more hardware it is possible to try to predict branch behavior during program execution.

One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called **dynamic branch prediction**.

One implementation of that approach is a **branch prediction buffer** or **branch history table**. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

This is the simplest sort of buffer; we don't know, in fact, if the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. But this doesn't affect correctness. Prediction is just a hint that is assumed to be correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

dynamic branch prediction

Prediction of branches at runtime using runtime information.

branch prediction buffer

Also called **branch history table**. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

Loops and Prediction

Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

EXAMPLE

ANSWER

The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will say taken: the branch has been taken nine times in a row at that point. The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration. Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones).

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed. Figure 6.39 shows the finite state machine for a 2-bit prediction scheme.

A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; as mentioned on page 418, it can be as early as the ID stage. Otherwise, sequential

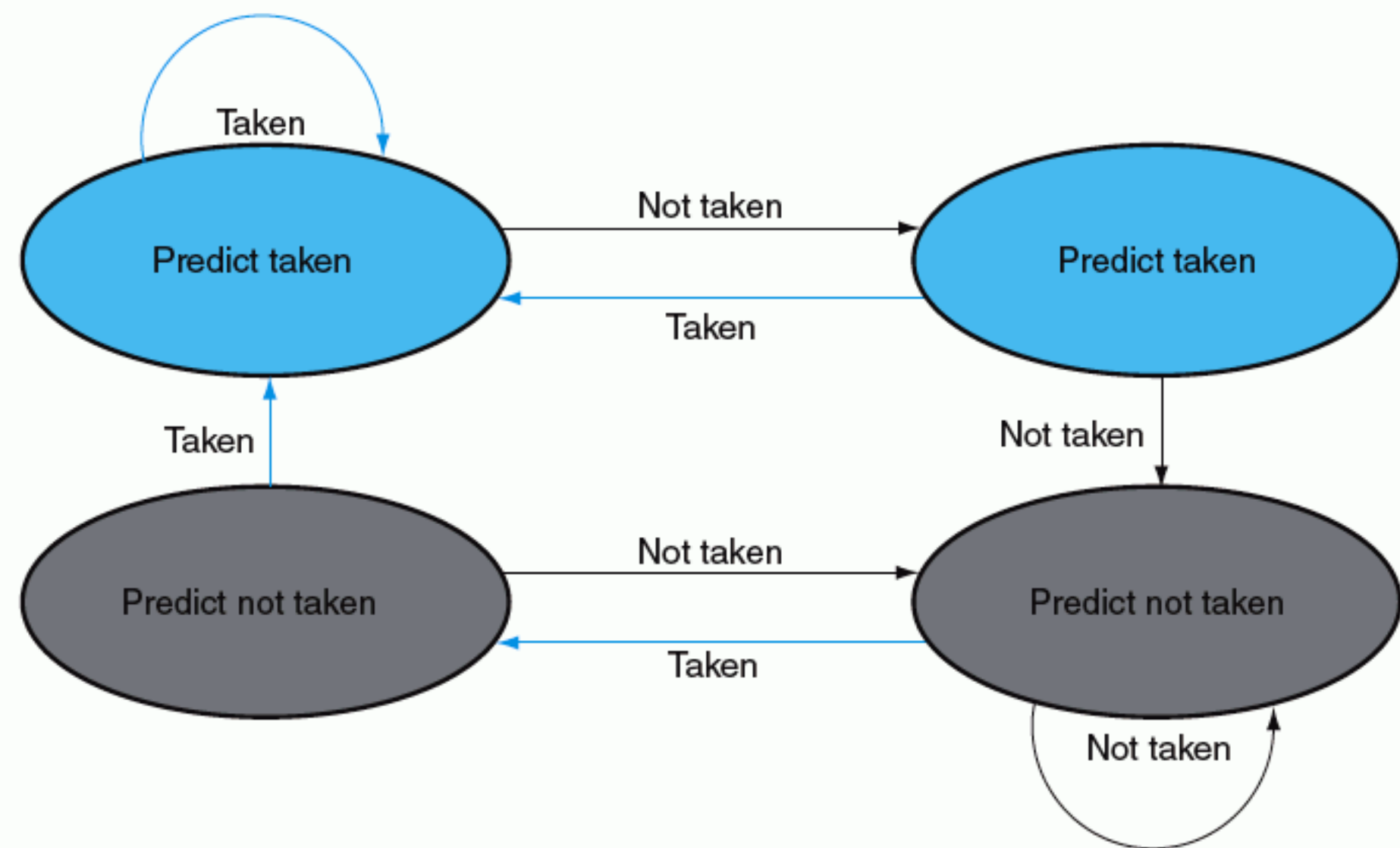


FIGURE 6.39 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The two-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the midpoint of its range as the division between taken and not taken.

fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 6.39.

Elaboration: As we described in Section 6.1, in a five-stage pipeline we can make the control hazard a feature by redefining the branch. A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch.

Compilers and assemblers try to place an instruction that always executes after the branch in the **branch delay slot**. The job of the software is to make the successor instructions valid and useful. Figure 6.40 shows the three ways in which the branch delay slot can be scheduled.

The limitations on delayed-branch scheduling arise from (1) the restrictions on the instructions that are scheduled into the delay slots and (2) our ability to predict at compile time whether a branch is likely to be taken or not.

Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle. As processors go to both longer pipelines and issuing multiple instructions per clock cycle (see Section 6.9), the branch delay becomes longer and a single delay slot is insufficient. Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches. Simultaneously, the growth in available transistors per chip has made dynamic prediction relatively cheaper.

Elaboration: A branch predictor tells us whether or not a branch is taken, but still requires the calculation of the branch target. In the five-stage pipeline, this calculation takes 1 cycle, meaning that taken branches will have a 1-cycle penalty. Delayed branches are one approach to eliminate that penalty. Another approach is to use a cache to hold the destination program counter or destination instruction, using a **branch target buffer**.

Elaboration: The 2-bit dynamic prediction scheme uses only information about a particular branch. Researchers noticed that using information about both a local branch and the global behavior of recently executed branches together yields greater prediction accuracy for the same number of prediction bits. Such predictors are called **correlating predictors**. A typical correlating predictor might have two 2-bit predictors for each branch with the choice between predictors made on the basis of whether the last executed branch was taken or not taken. Thus, the global branch behavior can be thought of as adding additional index bits for the prediction lookup.

A more recent innovation in branch prediction is the use of tournament predictors. A **tournament predictor** uses multiple predictors, tracking, for each branch, which predictor yields the best results. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behavior. A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor favoring whichever of the two predictors has been more accurate. Many recent advanced microprocessors make use of such elaborate predictors.

branch delay slot The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

branch target buffer A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

correlating predictor A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

tournament branch predictor A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

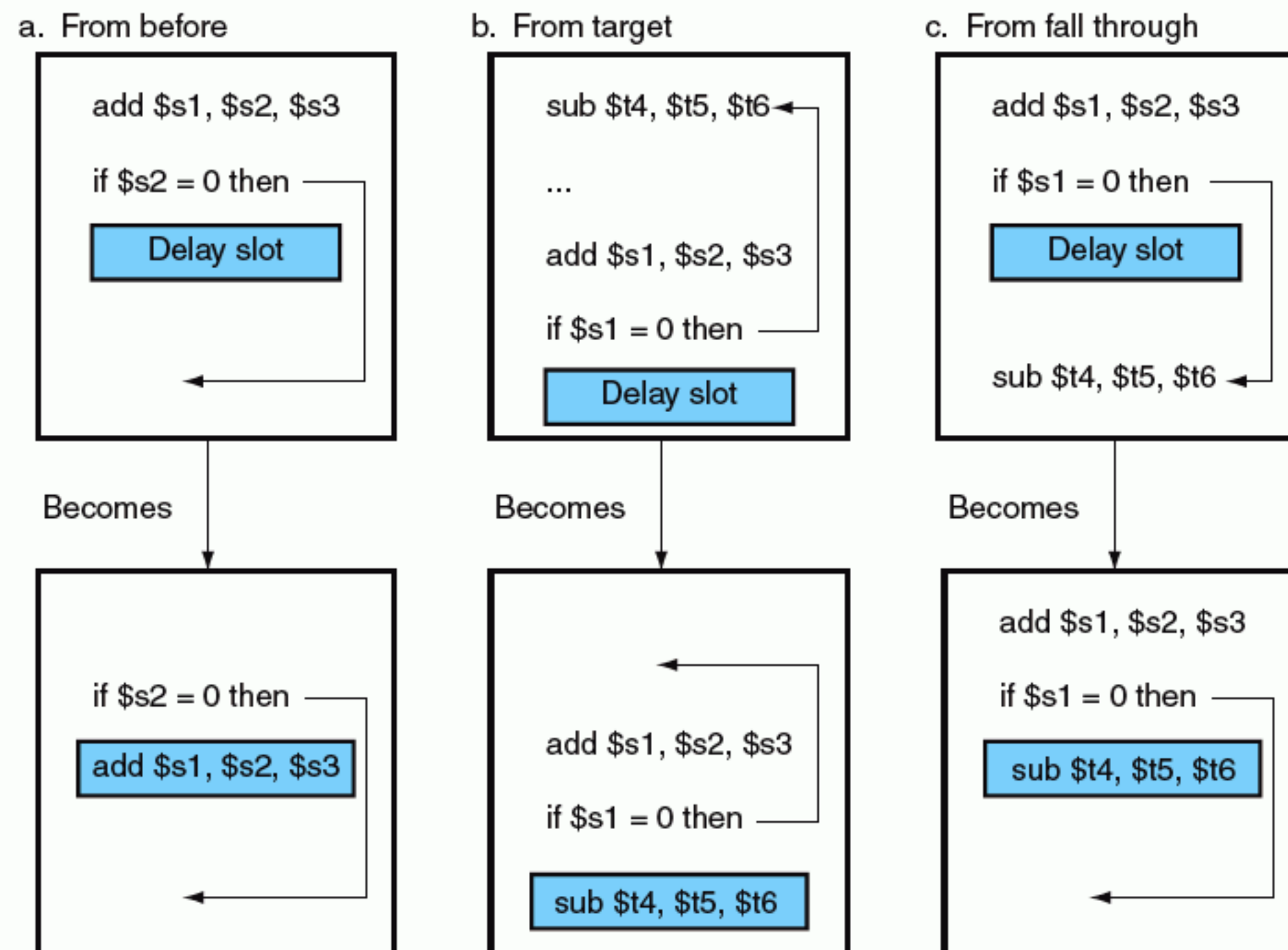


FIGURE 6.40 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `$s1` in the branch condition prevents the `add` instruction (whose destination is `$s1`) from being moved into the branch delay slot. In (b) the branch-delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By “OK” we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `$t4` were an unused temporary register when the branch goes in the unexpected direction.

Pipeline Summary

Thus far, we have seen three models of execution: single cycle, multicycle, and pipelined. Pipelined control strives for 1 clock cycle per instruction, like single cycle, but also for a fast clock cycle, like multicycle. Let’s revisit the example comparison of single-cycle and multicycle processors.

Comparing Performance of Several Control Schemes

Compare performance for single-cycle, multicycle, and pipelined control using the SPECint2000 instruction mix (see examples on pages 315 and 330) and assuming the same cycle times per unit as the example on page 315. For pipelined execution, assume that half of the load instructions are immediately followed by an instruction that uses the result, that the branch delay on misprediction is 1 clock cycle, and that one-quarter of the branches are mispredicted. Assume that jumps always pay 1 full clock cycle of delay, so their average time is 2 clock cycles. Ignore any other hazards.

EXAMPLE

From the example on page 315 (Performance of Single-Cycle Machines), we get the following functional unit times:

- 200 ps for memory access
- 100 ps for ALU operation
- 50 ps for register file read or write

For the single-cycle datapath, this leads to a clock cycle of

$$200 + 50 + 100 + 200 + 50 = 600 \text{ ps}$$

The example on page 330 (CPI in a Multicycle CPU) has the following instruction frequencies:

- 25% loads
- 10% stores
- 11% branches
- 2% jumps
- 52% ALU instructions

Furthermore, the example on page 330 showed that the CPI for the multiple design was 4.12. The clock cycle for the multicycle datapath and the pipelined design must be the same as the longest functional unit: 200 ps.

For the pipelined design, loads take 1 clock cycle when there is no load-use dependence and 2 when there is. Hence, the average clock cycles per load instruction is 1.5. Stores take 1 clock cycle, as do the ALU instructions. Branches take 1 when predicted correctly and 2 when not, so the average clock cycles per branch instruction is 1.25. The jump CPI is 2. Hence the average CPI is

$$1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$$

ANSWER

Let's compare the three designs by the average instruction time. For the single-cycle design, it is fixed at 600 ps. For the multicycle design, it is $200 \times 4.12 = 824$ ps. For the pipelined design, the average instruction time is $1.17 \times 200 = 234$ ps, making it almost twice as fast as either approach.

The clever reader will notice that the long cycle time of the memory is a performance bottleneck for both the pipelined and multicycle designs. Breaking memory accesses into two clock cycles and thereby allowing the clock cycle to be 100 ps would improve the performance in both cases. We explore this in the exercises.

This chapter started in the laundry room, showing principles of pipelining in an everyday setting. Using that analogy as a guide, we explained instruction pipelining step-by-step, starting with the single-cycle datapath and then adding pipeline registers, forwarding paths, data hazard detection, branch prediction, and flushing instructions on exceptions. Figure 6.41 shows the final evolved datapath and control.

Check Yourself

Consider three branch prediction schemes: branch not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and 2 cycles when they are wrong. Assume that the average predict accuracy of the dynamic predictor is 90%. Which predictor is the best choice for the following branches?

1. A branch that is taken with 5% frequency
2. A branch that is taken with 95% frequency
3. A branch that is taken with 70% frequency



Using a Hardware Description Language to Describe and Model a Pipeline

This section, which appears on the CD, provides a behavioral model in Verilog of the MIPS five-stage pipeline. The initial model ignores hazards, and additions to the model highlight the changes for forwarding, data hazards, and branch hazards.

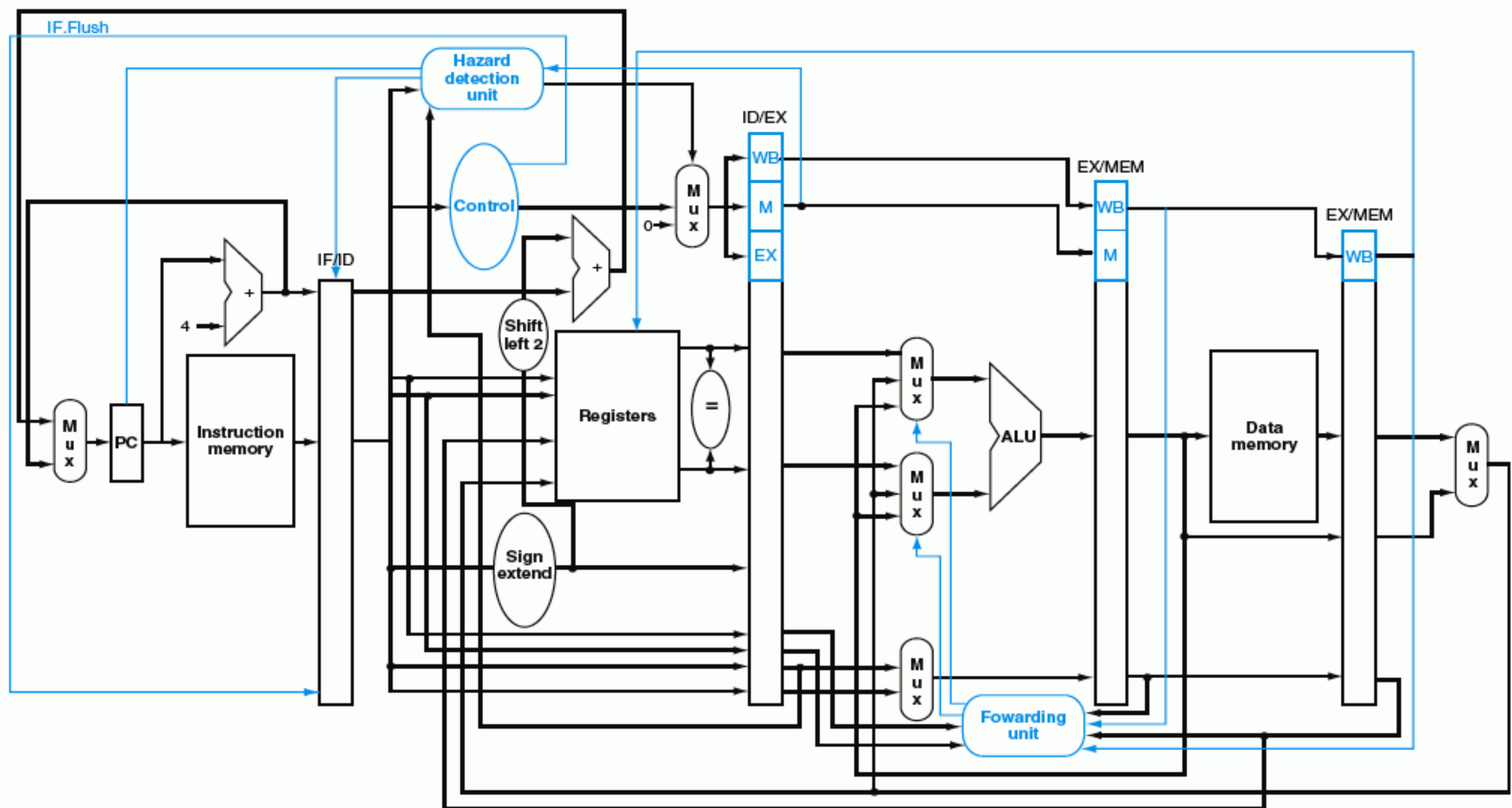


FIGURE 6.41 The final datapath and control for this chapter.

6.8 Exceptions

Another form of control hazard involves exceptions. For example, suppose the following instruction

```
add $1,$2,$1
```

has an arithmetic overflow. We need to transfer control to the exception routine immediately after this instruction because we wouldn't want this invalid value to contaminate other registers or memory locations.

Just as we did for the taken branch in the previous section, we must flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address. We will use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines.

When we dealt with branch mispredict, we saw how to flush the instruction in the IF stage by turning it into a nop. To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new

To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

control signal, called ID.Flush, is ORed with the stall signal from the Hazard Detection Unit to flush during ID. To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexors to zero the control lines. To start fetching instructions from location $8000\ 0180_{\text{hex}}$, which is the exception location for an arithmetic overflow, we simply add an additional input to the PC multiplexor that sends $8000\ 0180_{\text{hex}}$ to the PC. Figure 6.42 shows these changes.

This example points out a problem with exceptions: If we do not stop execution in the middle of the instruction, the programmer will not be able to see the original value of register \$1 that helped cause the overflow because it will be clobbered as the destination register of the add instruction. Because of careful planning, the overflow exception is detected during the EX stage; hence, we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. Many exceptions require that we eventually complete the instruction that caused the exception as if it executed normally. The easiest way to do this

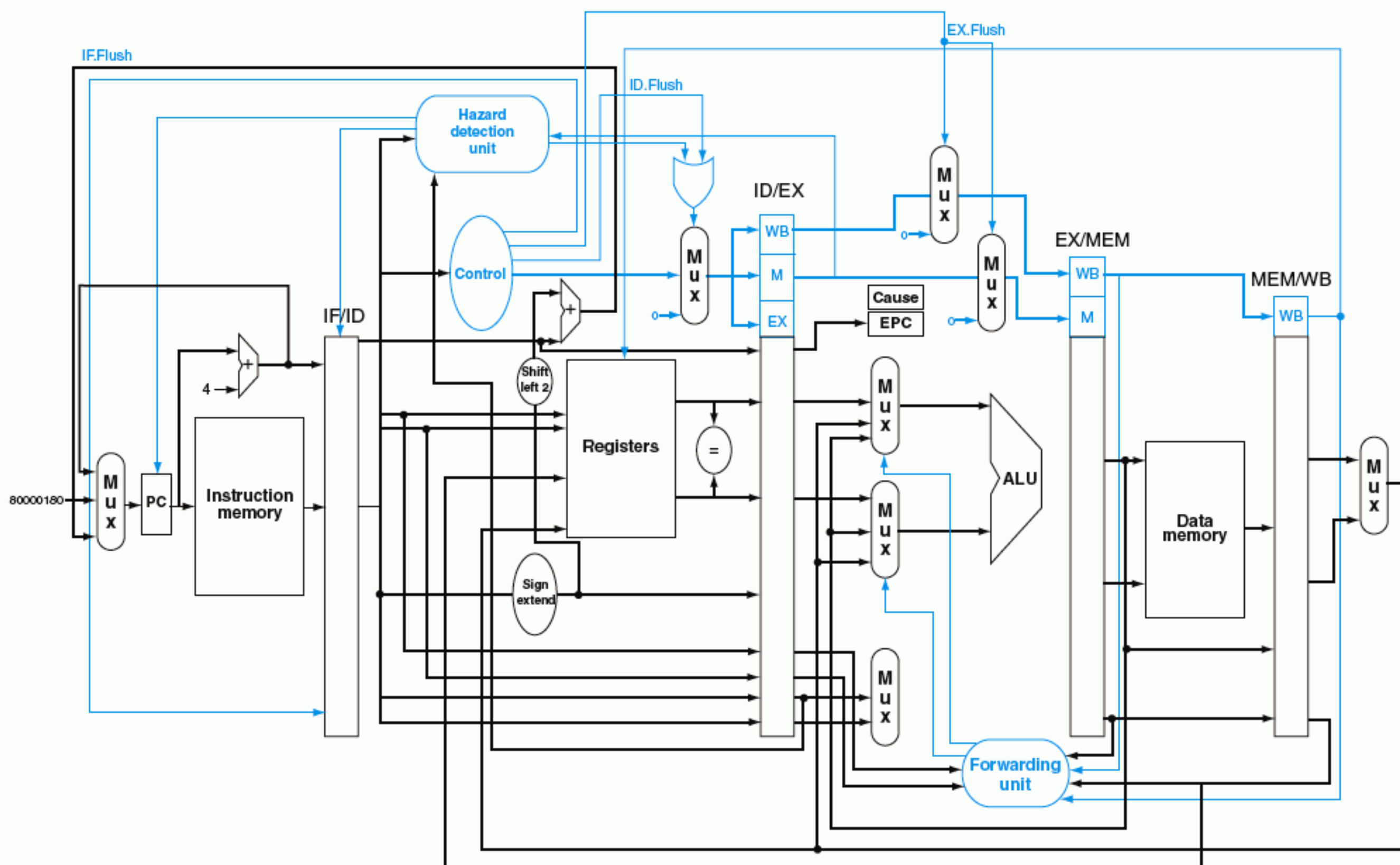


FIGURE 6.42 The datapath with controls to handle exceptions. The key additions include a new input, with the value $8000\ 0180_{\text{hex}}$, in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The $8000\ 0180_{\text{hex}}$ input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

is to flush the instruction and restart it from the beginning after the exception is handled.

The final step is to save the address of the offending instruction in the Exception Program Counter (EPC), as we did in Chapter 5. In reality, we save the address + 4, so the exception handling routine must first subtract 4 from the saved value. Figure 6.42 shows a stylized version of the datapath, including the branch hardware and necessary accommodations to handle exceptions.

Exception in a Pipelined Computer

Given this instruction sequence,

```
40hex    sub    $11, $2, $4
44hex    and    $12, $2, $5
48hex    or     $13, $2, $6
4Chex    add    $1,  $2, $1
50hex    slt    $15, $6, $7
54hex    lw     $16, 50($7)
...
```

assume the instructions to be invoked on an exception begin like this:

```
40000040hex    SW      $25, 1000($0)
40000044hex    SW      $26, 1004($0)
...
```

Show what happens in the pipeline if an overflow exception occurs in the add instruction.

Figure 6.43 shows the events, starting with the add instruction in the EX stage. The overflow is detected during that phase, and 4000 0040_{hex} is forced into the PC. Clock cycle 7 shows that the add and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction *following* the add is saved: 4C_{hex} + 4 = 50_{hex}.

EXAMPLE

ANSWER

Chapter 5 lists some other causes of exceptions:

- I/O device request
- Invoking an operating system service from a user program
- Using an undefined instruction
- Hardware malfunction

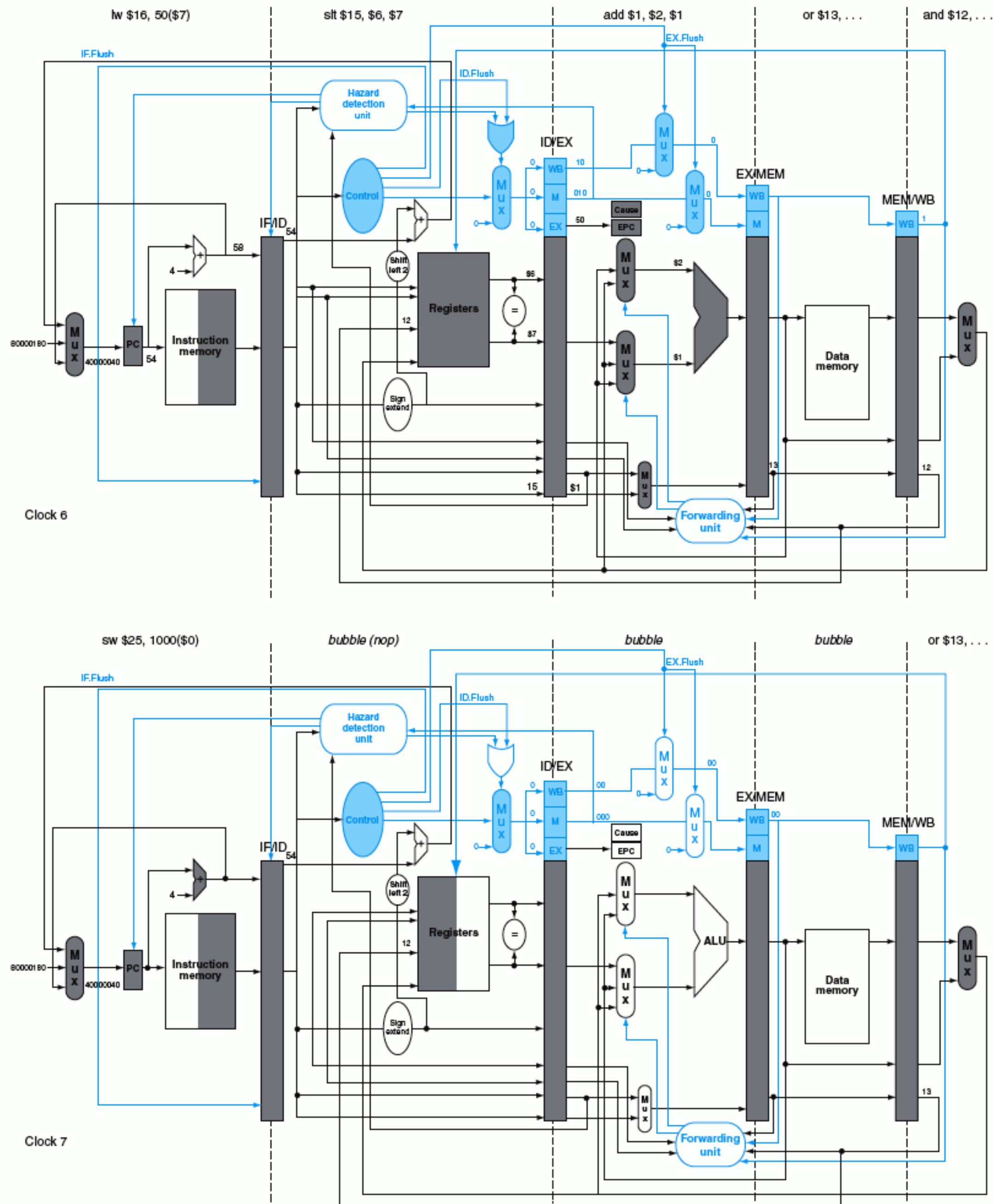


FIGURE 6.43 The result of an exception due to arithmetic overflow in the add instruction. The overflow is detected during the EX stage of clock 6, saving the address following the add in the EPC register ($4C + 4 = 50_{\text{hex}}$). Overflow causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the add. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—`sw $25, 1000($0)`—from instruction location $4000\ 0040_{\text{hex}}$. Note that the `and` and `or` instructions, which are prior to the add, still complete. Although not shown, the ALU overflow signal is an input to the control unit.

With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction. Moreover, multiple exceptions can occur simultaneously in a single clock cycle. The normal solution is to prioritize the exceptions so that it is easy to determine which is serviced first; this strategy works for pipelined processors as well. In most MIPS implementations, the hardware sorts exceptions so that the earliest instruction is interrupted.

I/O device requests and hardware malfunctions are not associated with a specific instruction, so the implementation has some flexibility as to when to interrupt the pipeline. Hence, using the mechanism used for other exceptions works just fine.

The EPC captures the address of the interrupted instructions, and the MIPS Cause register records all possible exceptions in a clock cycle, so the exception software must match the exception to the instruction. An important clue is knowing in which pipeline stage a type of exception can occur. For example, an undefined instruction is discovered in the ID stage, and invoking the operating system occurs in the EX stage. Exceptions are collected in the Cause register so that the hardware can interrupt based on later exceptions, once the earliest one has been serviced.

The hardware and the operating system must work in conjunction so that exceptions behave as you would expect. The hardware contract is normally to stop the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address. The operating system contract is to look at the cause of the exception and act appropriately. For an undefined instruction, hardware failure, or arithmetic overflow exception, the operating system normally kills the program and returns an indicator of the reason. For an I/O device request or an operating system service call, the operating system saves the state of the program, performs the desired task, and, at some point in the future, restores the program to continue execution. In the case of I/O device requests, we may often choose to run another task before resuming the task that requested the I/O, since that task may often not be able to proceed until the I/O is complete. This is why the ability to save and restore the state of any task is critical. One of the most important and frequent uses of exceptions is handling page faults and TLB exceptions; Chapter 7 describes these exceptions and their handling in more detail.

Hardware Software Interface

imprecise interrupt Also called imprecise exception. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

precise interrupt Also called **precise exception**. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

The difficulty of always associating the correct exception with the correct instruction in pipelined computers has led some computer designers to relax this requirement in noncritical cases. Such processors are said to have **imprecise interrupts** or **imprecise exceptions**. In the example above, PC would normally have 58_{hex} at the start of the clock cycle after the exception is detected, even though the offending instruction is at address $4C_{\text{hex}}$. A processor with imprecise exceptions might put 58_{hex} into EPC and leave it up to the operating system to determine which instruction caused the problem. MIPS and the vast majority of computers today support **precise interrupts** or **precise exceptions**. (One reason is to support virtual memory, which we shall see in Chapter 7.)

Check Yourself

The MIPS designers wanted the integer multiply and divide instructions to operate in parallel with other integer instructions. Since multiply and divide take multiple clock cycles, a group of students is arguing over whether it is possible to implement precise exceptions. Which of the following arguments are completely accurate?

1. It is impossible to implement precise exceptions, since a multiply or divide can raise an exception after instructions that follow it.
2. It is trivial to implement precise exceptions since multiply and divide cannot raise an exception once they start, and so the timing of all exceptions is obviously precise.
3. It does not matter whether multiply or divide can raise an exception. The fact that they could still be executing and not completed when some other instruction raised an exception makes it impossible to implement precise exceptions.
4. Although it is true that a multiply or divide could still be executing, it is guaranteed to complete shortly, and when it does, any exception raised for an instruction following a multiply or divide will then be precise.

6.9

Advanced Pipelining: Extracting More Performance

Be forewarned that Sections 6.9 and 6.10 are brief overviews of fascinating but advanced topics. If you want to learn more details, you should consult our more

advanced book, *Computer Architecture: A Quantitative Approach*, third edition, where the material covered in the next 18 pages is expanded to over 200 pages!

Pipelining exploits the potential parallelism among instructions. This parallelism is called **instruction-level parallelism** (ILP). There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Using our laundry analogy and assuming that the washer cycle were longer than the others, we could divide our washer into three machines that perform the wash, rinse, and spin steps of a traditional washer. We would then move from a four-stage to a six-stage pipeline. To get the full speedup, we need to rebalance the remaining steps so they are the same length, in processors or in laundry. The amount of parallelism being exploited is higher, since there are more operations being overlapped. Performance is potentially greater since the clock cycle can be shorter.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, for the CPI to be less than 1. It is sometimes useful to flip the metric, and use *IPC*, or *instructions per clock cycle*, particularly as values become less than 1! Hence, a 6 GHz four-way multiple-issue microprocessor can execute a peak rate of 24 billion instructions per second and have a best case CPI of 0.25, or IPC of 4. Assuming a five-stage pipeline, such a processor would have 20 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to eight instructions in every clock cycle. There are typically, however, many constraints on what types of instructions may be executed simultaneously and what happens when dependencies arise.

There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called **static multiple issue** and **dynamic multiple issue**. As we will see, both approaches have other, more commonly used names, which may be less precise or more restrictive.

instruction-level parallelism The parallelism among instructions.

multiple issue A scheme whereby multiple instructions are launched in 1 clock cycle.

static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

dynamic multiple issue An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

issue slots The positions from which instructions could issue in a given clock cycle; by analogy these correspond to positions at the starting blocks for a sprint.

There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

1. Packaging instructions into **issue slots**: How does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.
2. Dealing with data and control hazards: In static issue processors, some or all of the consequences of data and control hazards are handled statically by the compiler. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

Although we describe these as distinct approaches, in reality techniques from one approach are often borrowed by the other, and neither approach can claim to be perfectly pure.

The Concept of Speculation

speculation An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

One of the most important methods for finding and exploiting more ILP is speculation. **Speculation** is an approach that allows the compiler or the processor to “guess” about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction. For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier. Or, we might speculate that a store that precedes a load does not refer to the same address, which would allow the load to be executed before the store. The difficulty with speculation is that it may be wrong. So, any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively. The implementation of this back-out capability adds complexity to any processor supporting speculation.

Speculation may be done in the compiler or by the hardware. For example, the compiler can use speculation to reorder instructions, moving an instruction across a branch or a load across a store. The processor hardware can perform the same transformation at runtime using techniques we discuss later in this section.

The recovery mechanisms used for incorrect speculation are rather different. In the case of speculation in software, the compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to

use when the speculation was incorrect. In hardware speculation, the processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation was correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation was incorrect, the hardware flushes the buffers and reexecutes the correct instruction sequence.

Speculation introduces one other possible problem: speculating on certain instructions may introduce exceptions that were formerly not present. For example, suppose a load instruction is moved in a speculative manner, but the address it uses is not legal when the speculation is incorrect. The result would be that an exception that should not have occurred will occur. The problem is complicated by the fact that if the load instruction were not speculative, then the exception must occur! In compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point the exception is raised, and normal exception handling proceeds.

Since speculation can improve performance when done properly and decrease performance when done carelessly, significant effort goes into deciding when it is appropriate to speculate. Later in this section, we will examine both static and dynamic techniques for speculation.

Static Multiple Issue

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions that issue in a given clock cycle, which is called an **issue packet**, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: Very Long Instruction Word (VLIW). The Intel IA-64 architecture uses this approach, which it calls by its own name: Explicitly Parallel Instruction Computer (EPIC). The Itanium and Itanium 2 processors, available in 2000 and 2002, respectively, are the first implementations of the IA-64 architecture.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards.

issue packet The set of instructions that issues together in 1 clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Let’s look at a simple static issue version of a MIPS processor, before we describe the use of these techniques in more aggressive processors. After using this simple example to review the comments, we discuss the highlights of the Intel IA-64 architecture.

An Example: Static Multiple Issue with the MIPS ISA

To give a flavor of static multiple issue, we consider a simple two-issue MIPS processor, where one of the instructions can be an integer ALU operation or branch, and the other can be a load or store. Such a design is like that used in some embedded MIPS processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue. Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a no-op. Thus, the instructions always issue in pairs, possibly with a nop in one slot. Figure 6.44 shows how the instructions look as they go into the pipeline in pairs.

Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls. In others, the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction pair. Even so, a hazard generally forces the entire issue packet containing the dependent instruction to stall. Whether the software must handle all hazards or

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 6.44 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

only try to reduce the fraction of hazards between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced. We will assume the second approach for this example.

To issue an ALU and a data transfer operation in parallel, the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see Figure 6.45). In 1 clock cycle we may need to read two registers for the ALU operation and two more for a store, and also one write port for an ALU operation and one write port for a load. Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

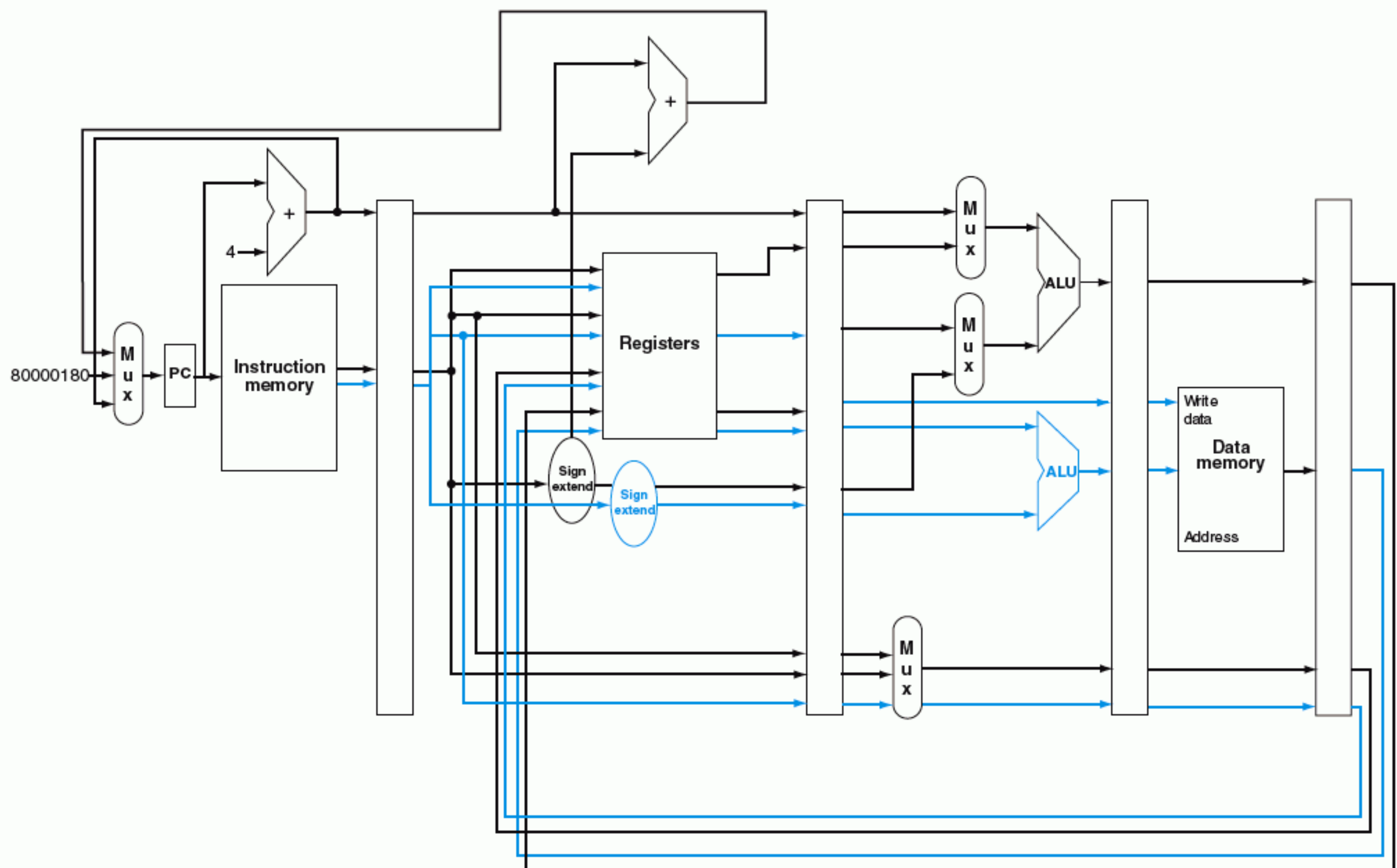


FIGURE 6.45 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Clearly, this two-issue processor can improve performance by up to a factor of 2. Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards. For example, in our simple five-stage pipeline, loads have a use latency of 1 clock cycle, which prevents one instruction from using the result without stalling. In the two-issue, five-stage pipeline, the result of a load instruction cannot be used on the next *clock cycle*. This means that the next *two* instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline, now have a one-instruction use latency, since the results cannot be used in the paired load or store. To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler takes on this role.

EXAMPLE

Simple Multiple-Issue Code Scheduling

How would this loop be scheduled on a static two-issue pipeline for MIPS?

```

Loop:   lw      $t0, 0($s1)      # $t0=array element
        addu    $t0,$t0,$s2      # add scalar in $s2
        sw      $t0, 0($s1)      # store result
        addi    $s1,$s1,-4       # decrement pointer
        bne     $s1,$zero,Loop   # branch $s1!=0

```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

ANSWER

The first three instructions have data dependences, and so do the last two. Figure 6.46 shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes 4 clocks per loop iteration; at 4 clocks to execute 5 instructions, we get the disappointing CPI of 0.8 versus the best case of 0.5, or an IPC of 1.25 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

An important compiler technique to get more performance from loops is **loop unrolling**, a technique where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1,\$s1,-4		2
	addu \$t0,\$t0,\$s2		3
	bne \$s1,\$zero,Loop	sw \$t0, 4(\$s1)	4

FIGURE 6.46 The scheduled code as it would look on a two-issue MIPS pipeline. The empty slots are nops.

Loop Unrolling for Multiple-Issue Pipelines

See how well loop unrolling and scheduling work in the example above. Assume that the loop index is a multiple of four, for simplicity.

To schedule the loop without any delays, it turns out that we need to make four copies of the loop body. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of lw, add, and sw, plus one addi and one bne. Figure 6.47 shows the unrolled and scheduled code.

During the unrolling process, the compiler introduced additional registers (\$t1, \$t2, \$t3). The goal of this process, called **register renaming**, is to eliminate dependences that are not true data dependences, but could either lead to potential hazards or prevent the compiler from flexibly scheduling the code. Consider how the unrolled code would look using only \$t0. There would be repeated instances of lw \$t0,0(\$s1), addu \$t0,\$t0,\$s2 followed by sw \$t0,4(\$s1), but these sequences, despite using \$t0, are actually completely independent—no data values flow between one pair of these instructions and the next pair. This is what is called an **antidependence** or **name dependence**, which is an ordering forced purely by the reuse of a name, rather than a real data dependence.

Renaming the registers during the unrolling process allows the compiler to subsequently move these independent instructions so as to better schedule the code. The renaming process eliminates the name dependences, while preserving the true dependences.

Notice now that 12 of the 14 instructions in the loop execute as a pair. It takes 8 clocks for four loop iterations, or 2 clocks per iteration, which yields a CPI of $8/14 = 0.57$. Loop unrolling and scheduling with dual issue gave us a factor of two improvement, partly from reducing the loop control instructions and partly from dual issue execution. The cost of this performance improvement is using four temporary registers rather than one, as well as a significant increase in code size.

EXAMPLE

ANSWER

register renaming The renaming of registers, by the compiler or hardware, to remove antidependences.

antidependence Also called **name dependence**. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi \$s1,\$s1,-16	lw \$t0, 0(\$s1)	1
		lw \$t1,12(\$s1)	2
	addu \$t0,\$t0,\$s2	lw \$t2, 8(\$s1)	3
	addu \$t1,\$t1,\$s2	lw \$t3, 4(\$s1)	4
	addu \$t2,\$t2,\$s2	sw \$t0, 16(\$s1)	5
	addu \$t3,\$t3,\$s2	sw \$t1,12(\$s1)	6
		sw \$t2, 8(\$s1)	7
	bne \$s1,\$zero,Loop	sw \$t3, 4(\$s1)	8

FIGURE 6.47 The unrolled and scheduled code of Figure 6.46 as it would look on a static two-issue MIPS pipeline. The empty slots are nops. Since the first instruction in the loop decrements \$s1 by 16, the addresses loaded are the original value of \$s1, then that address minus 4, minus 8, and minus 12.

The Intel IA-64 Architecture

The IA-64 architecture is a register-register, RISC-style instruction set like the 64-bit version of the MIPS architecture (called MIPS-64), but with several unique features to support explicit, compiler-driven exploitation of ILP. Intel calls the approach EPIC (Explicitly Parallel Instruction Computer). The major differences between IA-64 and the MIPS architecture are the following:

- 1. IA-64 has many more registers than MIPS, including 128 integer and 128 floating-point registers, as well as 8 special registers for branches and 64 1-bit condition registers. In addition, IA-64 supports register windows in a fashion similar to the original Berkeley RISC and Sun SPARC architectures.
- 2. IA-64 places instructions into bundles that have a fixed format and explicit designation of dependences.
- 3. IA-64 includes special instructions and capabilities for speculation and for branch elimination, which increase the amount of ILP that can be exploited.

The IA-64 architecture is designed to achieve the major benefits of a VLIW—implicit parallelism among operations in an instruction and fixed formatting of the operation fields—while maintaining greater flexibility than a VLIW normally allows. The IA-64 architecture uses two different concepts to achieve this flexibility: instruction groups and bundles.

An **instruction group** is a sequence of consecutive instructions with no register data dependences among them. All the instructions in a group could be executed in parallel if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must *explicitly* indicate the boundary between one instruction group and another. This boundary is indicated by placing a **stop** between two instructions that belong to different groups.

instruction group In IA-64, a sequence of consecutive instructions with no register data dependences among them.

stop In IA-64, an explicit indicator of a break between independent and dependent instructions.

IA-64 instructions are encoded in *bundles*, which are 128 bits wide. Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length. To simplify the decoding and instruction issue process, the template field of a bundle specifies which of five different execution units each instruction in the bundle requires. The five different execution units are integer ALU, noninteger ALU (includes shifters and multimedia operations), memory unit, floating-point unit, and branch unit.

The 5-bit template field within each bundle describes *both* the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. The bundle formats can specify only a subset of all possible combinations of instruction types and stops.

To enhance the amount of ILP that can be exploited, IA-64 provides extensive support for predication and for speculation (see the Elaboration on page 442). **Predication** is a technique that can be used to eliminate branches by making the execution of an instruction dependent on a predicate, rather than dependent on a branch. As we saw earlier, branches reduce the opportunity to exploit ILP by restricting the movement of code. Loop unrolling works well to eliminate loop branches, but a branch within a loop—arising, for example, from an *if-then-else* statement—cannot be eliminated by loop unrolling. Predication, however, provides a method to eliminate the branch, allowing more flexible exploitation of parallelism.

For example, suppose we had a code sequence like

```
if (p) {statement 1} else {statement 2}
```

Using normal compilation methods, this segment would compile using two branches: one after the condition branching to the else portion and one after statement 1 branching to the next sequential statement. With predication, it could be compiled as

```
(p) statement 1  
(~p) statement 2
```

where the use of (condition) indicates that the statement is executed only if condition is true, and otherwise becomes a no-op. Notice that predication can be used as way to speculate, as well as a method to eliminate branches.

The IA-64 architecture provides comprehensive support for predication: nearly every instruction in the IA-64 architecture can be predicated by specifying a predicate register, whose identity is placed in the lower 6 bits of an instruction field. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!

IA-64 is the most sophisticated example of an instruction set with support for compiler-based exploitation of ILP. Intel's Itanium and Itanium 2 processors implement this architecture. A brief summary of the characteristics of these processors is given in Figure 6.48.

predication A technique to make instructions dependent on predicates rather than on branches.

Processor	Maximum instr. issues / clock	Functional units	Maximum ops. per clock	Max. clock rate	Transistors (millions)	Power (watts)	SPEC int2000	SPEC fp2000
Itanium	6	4 integer/media 2 memory 3 branch 2 FP	9	0.8 GHz	25	130	379	701
Itanium 2	6	6 integer/media 4 memory 3 branch 2 FP	11	1.5 Ghz	221	130	810	1427

FIGURE 6.48 A summary of the characteristics of the Itanium and Itanium 2, Intel’s first two implementations of the IA-64 architecture. In addition to higher clock rates and more functional units, the Itanium 2 includes an on-chip level 3 cache, versus an off-chip level 3 cache in the Itanium.

poison A result generated when a speculative load yields an exception, or an instruction uses a poisoned operand.

advanced load In IA-64, a speculative load instruction with support to check for aliases that could invalidate the load.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

Elaboration: Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exceptions for speculated instructions, and memory reference speculation, which supports speculation of load instructions. Deferred exception handling is supported by adding speculative load instructions, which, when an exception occurs, tag the result as **poison**. When a poisoned result is used by an instruction, the result is also poison. The software can then check for a poisoned result when it knows that the execution is no longer speculative.

In IA-64, we can also speculate on memory references by moving loads earlier than stores on which they may depend. This is done with an advanced load instruction. An **advanced load** executes normally, but uses a special table to track the address that the processor loaded from. All subsequent stores check that table and generate a flag in the entry if the store address matches the load address. A subsequent instruction must be used to check the status of the entry after the load is no longer speculative. If a store to the same address has intervened, the check instruction specifies a fix-up routine that reexecutes the load and any other dependent instructions before continuing execution; if no such store has occurred, the table entry is simply cleared, indicating that the load is no longer speculative.

Dynamic Multiple-Issue Processors

Dynamic multiple-issue processors are also known as **superscalar** processors, or simply superscalars. In the simplest superscalar processors, instructions issue in-order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly

independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

Many superscalars extend the basic framework of dynamic issue decisions to include **dynamic pipeline scheduling**. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub      $s4, $s4, $t3
slti    $t5, $s4, 20
```

Even though the `sub` instruction is ready to execute, it must wait for the `lw` and `addu` to complete first, which might take many clock cycles if memory is slow. (Chapter 7 explains caches, the reason that memory accesses are sometimes very slow.) Dynamic pipeline scheduling allows such hazards to be avoided either fully or partially.

Dynamic Pipeline Scheduling

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units (10 or more in high-end designs in 2004), and a **commit unit**. Figure 6.49 shows the model. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called **reservation stations**, that hold the operands and the operation. (In the next section, we will discuss an alternative to reservation stations used by many recent processors.) As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming, just like that used by

dynamic pipeline scheduling Hardware support for reordering the order of instruction execution so as to avoid stalls.

commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

reservation station A buffer within a functional unit that holds the operands and the operation.

reorder buffer The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

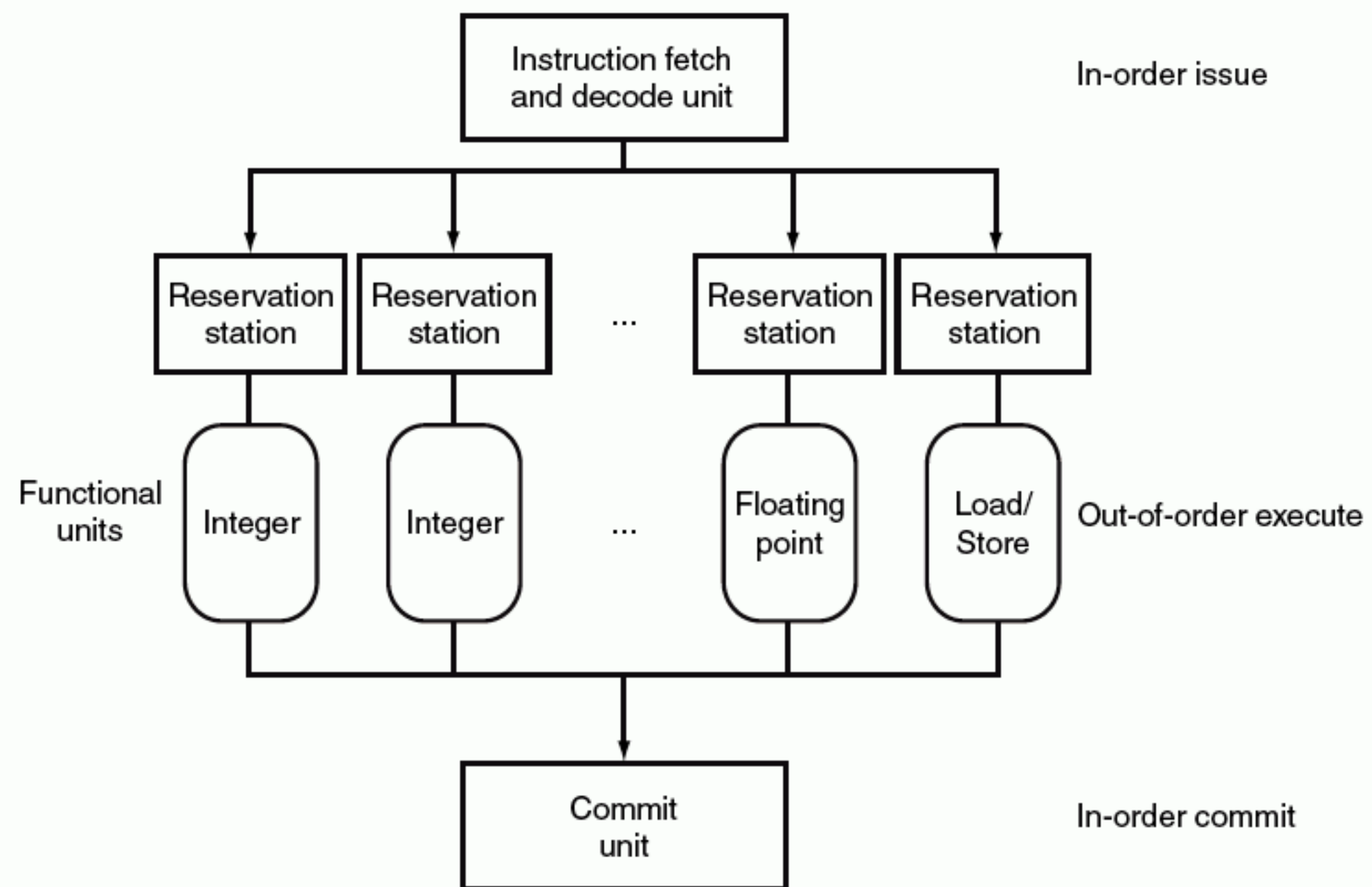


FIGURE 6.49 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

the compiler in our earlier loop unrolling example on page 439. To see how this conceptually works, consider the following steps:

1. When an instruction issues, if either of its operands is in the register file or the reorder buffer, it is copied to the reservation station immediately, where it is buffered until all the operands and an execution unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

Conceptually, you can think of a dynamically scheduled pipeline as analyzing the dataflow structure of a program, as we saw when we discussed dataflow analysis within a compiler in Chapter 2. The processor then executes the instructions in some order that preserves the data flow order of the program. To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program execution order. This conservative mode is called in-order completion. Hence, if an exception occurs, the computer can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception. Although, the front end (fetch and issue) and the back end (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the data they need is available. Today, all dynamically scheduled pipelines use in-order completion, although this was not always true.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path. Because the instructions are **committed in order**, we know whether or not the branch was correctly predicted before any instructions from the predicted path are committed. A speculative, dynamically scheduled pipeline can also support speculation on load addresses, allowing load-store reordering, and using the commit unit to avoid incorrect speculation. In the next section we will look at the use of dynamic scheduling with speculation in the Pentium 4 design.

Elaboration: A commit unit controls updates to the register file *and* memory. Some dynamically scheduled processors update the register file immediately during execution using extra registers to implement the renaming function and preserving the older copy of a register until the instruction updating the register is no longer speculative. Other processors buffer the result, typically in a structure called a reorder buffer, and the actual update to the register file occurs later as part of the commit. Stores to memory must be buffered until commit time either in a *store buffer* (see Chapter 7) or in the reorder buffer. The commit unit allows the store to write to memory from the buffer when the buffer has a valid address and valid data, and when the store is no longer dependent on predicted branches.

Elaboration: Memory accesses benefit from *nonblocking caches*, which continue servicing cache accesses during a cache miss (see Chapter 7). **Out-of-order execution** processors need nonblocking caches to allow instructions to execute during a miss.

in-order commit A commit in which the results of pipelined execution are written to the programmer-visible state in the same order that instructions are fetched.

out-of-order execution A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

Hardware Software Interface

Given that compilers can also schedule code around data dependences, you might ask, Why would a superscalar processor use dynamic scheduling? There are three major reasons. First, not all stalls are predictable. In particular, cache misses (see Chapter 7) cause unpredictable stalls. Dynamic scheduling allows the processor to hide some of those stalls by continuing to execute instructions while waiting for the stall to end.

Second, if the processor speculates on branch outcomes using dynamic branch prediction, it cannot know the exact order of instructions at compile time, since it depends on the predicted and actual behavior of branches. Incorporating dynamic speculation to exploit more ILP without incorporating dynamic scheduling would significantly restrict the benefits of such speculation.

Third, as the pipeline latency and issue width change from one implementation to another, the best way to compile a code sequence also changes. For example, how to schedule a sequence of dependent instructions is affected by both issue width and latency. The pipeline structure affects both the number of times a loop must be unrolled to avoid stalls as well as the process of compiler-based register renaming. Dynamic scheduling allows the hardware to hide most of these details. Thus, users and software distributors do not need to worry about having multiple versions of a program for different implementations of the same instruction set. Similarly, old legacy code will get much of the benefit of a new implementation without the need for recompilation.

The BIG Picture

Both pipelining and multiple-issue execution increase peak instruction throughput and attempt to exploit ILP. Data and control dependences in programs, however, offer an upper limit on sustained performance because the processor must sometimes wait for a dependence to be resolved. Software-centric approaches to exploiting ILP rely on the ability of the compiler to find and reduce the effects of such dependences, while hardware-centric approaches rely on extensions to the pipeline and issue mechanisms. Speculation, performed by the compiler or the hardware, can increase the amount of ILP that can be exploited, although care must be taken since speculating incorrectly is likely to reduce performance.

Modern, high-performance microprocessors are capable of issuing several instructions per clock; unfortunately, sustaining that issue rate is very difficult. For example, despite the existence of processors with four to six issues per clock, very few applications can sustain more than two instructions per clock. There are two primary reasons for this.

First, within the pipeline, the major performance bottlenecks arise from dependences that cannot be alleviated, thus reducing the parallelism among instructions and the sustained issue rate. Although little can be done about true data dependences, often the compiler or hardware does not know precisely whether a dependence exists or not, and so must conservatively assume the dependence exists. For example, code that makes use of pointers, particularly in ways that create more aliasing, will lead to more implied potential dependences. In contrast, the greater regularity of array accesses often allows a compiler to deduce that no dependences exist. Similarly, branches that cannot be accurately predicted whether at runtime or compile time will limit the ability to exploit ILP. Often additional ILP is available, but the ability of the compiler or the hardware to find ILP that may be widely separated (sometimes by the execution of thousands of instructions) is limited.

Second, losses in the memory system (the topic of Chapter 7) also limit the ability to keep the pipeline full. Some memory system stalls can be hidden, but limited amounts of ILP also limit the extent to which such stalls can be hidden.

State whether the following techniques or components are associated primarily with a software- or hardware-based approach to exploiting ILP. In some cases, the answer may be both.

1. Branch prediction
2. Multiple issue
3. VLIW
4. Superscalar
5. Dynamic scheduling
6. Out-of-order execution
7. Speculation
8. EPIC
9. Reorder buffer
10. Register renaming
11. Predication

Understanding Program Performance

Check Yourself

6.10 Real Stuff: The Pentium 4 Pipeline

In the last chapter, we discussed how the Pentium 4 fetched and translated IA-32 instructions into microoperations. The microoperations are then executed by a sophisticated, dynamically scheduled, speculative pipeline capable of sustaining an execution rate of three microoperations per clock cycle. This section focuses on that microoperation pipeline. The Pentium 4 combines multiple issue with deep pipelining so as to achieve both a low CPI and a high clock rate.

When we consider the design of sophisticated, dynamically scheduled processors, the design of the functional units, the cache and register file, instruction issue, and overall pipeline control become intermingled, making it difficult to separate out the datapath from the pipeline. Because of this, many engineers and researchers have adopted the term **microarchitecture** to refer to the detailed internal architecture of a processor. Figure 6.50 shows the microarchitecture of the Pentium 4, focusing on the structures for executing the microoperations.

microarchitecture The organization of the processor, including the major functional units, their interconnection, and control.

Another way to look at the Pentium 4 is to see the pipeline stages that a typical instruction goes through. Figure 6.51 shows the pipeline structure and the typical number of clock cycles spent in each; of course, the number of clock cycles varies due to the nature of dynamic scheduling as well as the requirements of individual microoperations.

The Pentium 4, and the earlier Pentium III and Pentium Pro, all use the technique of decoding IA-32 instructions into microoperations and executing those microoperations using a speculative pipeline with multiple functional units. In fact, the basic microarchitecture is similar, and all these processors can complete up to three microoperations per cycle. The Pentium 4 gains its performance advantage over the Pentium III through several enhancements:

1. A pipeline that is roughly twice as deep (approximately 20 cycles versus 10) and can run almost twice as fast in the same technology
2. More functional units (7 versus 5)
3. Support for a larger number of outstanding operations (126 versus 40)
4. The use of a trace cache (see Chapter 7) and a much better branch predictor (4K entries versus 512)
5. Other enhancements to the memory system, which we discuss in Chapter 7

architectural registers The instruction set visible registers of a processor; for example, in MIPS, these are the 32 integer and 16 floating-point registers.

Elaboration: The Pentium 4 uses a scheme for resolving antidependences and incorrect speculation that uses a reorder buffer together with register renaming. Register renaming explicitly renames the **architectural registers** in a processor (8 in the case of IA-32) to a larger set of physical registers (128 in the Pentium 4). The Pentium 4 uses

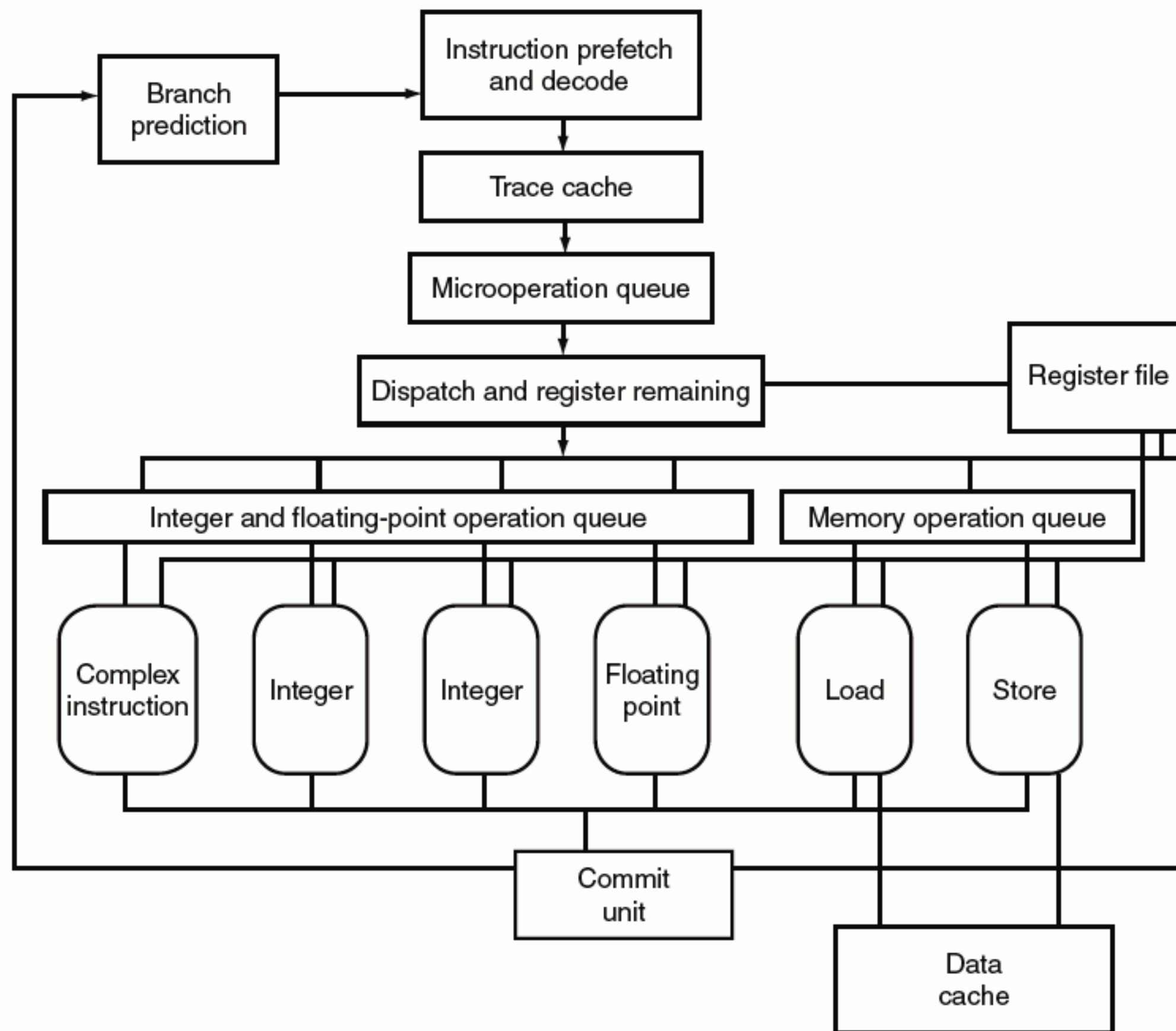


FIGURE 6.50 The microarchitecture of the Intel Pentium 4. The extensive queues allow up to 126 microoperations to be outstanding at any point in time, including 48 loads and 24 stores. There are actually seven functional units, since the FP unit includes a separate dedicated unit for floating-point moves. The load and store units are actually separated into two parts, with the first part handling address calculation and the second part responsible for the actual memory reference. The integer ALUs operate at twice the clock frequency, allowing two integer ALU operations to be completed by each of the two integer units in a single clock cycle. As we described in Chapter 5, the Pentium 4 uses a special cache, called the trace cache, to hold predecoded sequences of microoperations, corresponding to IA-32 instructions. The operation of a trace cache is explained in more detail in Chapter 7. The FP unit also handles the MMX multimedia and SSE2 instructions. There is an extensive bypass network among the functional units; since the pipeline is dynamic rather than static, bypassing is done by tagging results and tracking source operands, so as to allow a match when a result is produced for an instruction in one of the queues that needs the result. Intel is expected to release new versions of the Pentium 4 in 2004, which will probably have changes in the microarchitecture.

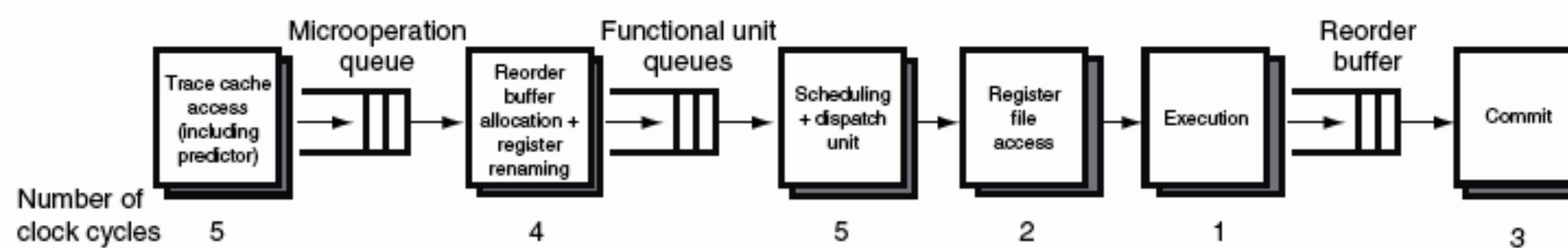


FIGURE 6.51 The Pentium 4 pipeline showing the pipeline flow for a typical instruction and the number of clock cycles for the major steps in the pipeline. The major buffers where instructions wait are also shown.

register renaming to remove antidependences. Register renaming requires the processor to maintain a map between the architectural registers and the physical registers, indicating which physical register is the most current copy of an architectural register. By keeping track of the renamings that have occurred, register renaming offers another approach to recovery in the event of incorrect speculation: simply undo the mappings that have occurred since the first incorrectly speculated instruction. This will cause the state of the processor to return to the last correctly executed instruction, keeping the correct mapping between the architectural and physical registers.

Understanding Program Performance

The Pentium 4 combines a deep pipeline (averaging 20 or more pipe stages per instruction) and aggressive multiple issue to achieve high performance. By keeping the latencies for back-to-back operations low (0 for ALU operations and 2 for loads), the impact of data dependences is reduced. What are the most serious potential performance bottlenecks for programs running on this processor? The following list includes some potential performance problems, the last three of which can apply in some form to any high-performance pipelined processor.

- The use of IA-32 instructions that do not map to three or fewer simple microoperations
- Branches that are difficult to predict, causing misprediction stalls and restarts when speculation fails
- Poor instruction locality, which causes the trace cache not to function effectively
- Long dependences—typically caused by long-running instructions or data cache misses—which lead to stalls

Performance delays arising in accessing memory (see Chapter 7) that cause the processor to stall

Check Yourself

Are the following statements true or false?

1. The Pentium 4 can issue more instructions per clock than the Pentium III.
2. The Pentium 4 multiple-issue pipeline directly executes IA-32 instructions.
3. The Pentium 4 uses dynamic scheduling but no speculation.
4. The Pentium 4 microarchitecture has many more registers than IA-32 requires.
5. The Pentium 4 pipeline has fewer stages than the Pentium III.
6. The trace cache in the Pentium 4 is exactly the same as an instruction cache.

6.11 Fallacies and Pitfalls

Fallacy: Pipelining is easy.

Our books testify to the subtlety of correct pipeline execution. Our advanced book had a pipeline bug in its first edition, despite its being reviewed by more than 100 people and being class-tested at 18 universities. The bug was uncovered only when someone tried to build the computer in that book. The fact that the Verilog to describe a pipeline like that in the Pentium 4 will be thousands of lines is an indication of the complexity. Beware!

Fallacy: Pipelining ideas can be implemented independent of technology.

When the number of transistors on-chip and speed of transistors made a five-stage pipeline the best solution, then the delayed branch (see the Elaboration on page 423) was a simple solution to control hazards. With longer pipelines, superscalar execution, and dynamic branch prediction, it is now redundant. In the early 1990s, dynamic pipeline scheduling took too many resources and was not required for high performance, but as transistor budgets continued to double and logic became much faster than memory, then multiple functional units and dynamic pipelining made more sense. Today, all high-end processors use multiple issue, and most choose to implement aggressive speculation as well.

Pitfall: Failure to consider instruction set design can adversely impact pipelining.

Many of the difficulties of pipelining arise because of instruction set complications. Here are some examples:

- Widely variable instruction lengths and running times can lead to imbalance among pipeline stages and severely complicate hazard detection in a design pipelined at the instruction set level. This problem was overcome, initially in the DEC VAX 8500 in the late 1980s, using the micropipelined scheme that the Pentium 4 employs today. Of course, the overhead of translation and maintaining correspondence between the microoperations and the actual instructions remains.
- Sophisticated addressing modes can lead to different sorts of problems. Addressing modes that update registers, such as update addressing (see Chapter 3), complicate hazard detection. Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.

Perhaps the best example is the DEC Alpha and the DEC NVAX. In comparable technology, the newer instruction set architecture of the Alpha allowed an implementation whose performance is more than twice as fast as NVAX. In another

example, Bhandarkar and Clark [1991] compared the MIPS M/2000 and the DEC VAX 8700 by counting clock cycles of the SPEC benchmarks; they concluded that, although the MIPS M/2000 executes more instructions, the VAX on average executes 2.7 times as many clock cycles, so the MIPS is faster.

Nine-tenths of wisdom consists of being wise in time.

American proverb

6.12

Concluding Remarks

Pipelining improves the average execution time per instruction. Depending on whether you start with a single-cycle or multiple-cycle datapath, this reduction can be thought of as decreasing the clock cycle time or as decreasing the number of clock cycles per instruction (CPI). We started with the simple single-cycle datapath, so pipelining was presented as reducing the clock cycle time of the simple datapath. Multiple issue, in comparison, clearly focuses on reducing CPI (or increasing IPC). Figure 6.52 shows the effect on CPI and clock rate for each of the microarchitectures from Chapters 5 and 6. Performance is increased by moving up and to the right, since it is the product of IPC and clock rate that determines performance for a given instruction set.

Pipelining improves throughput, but not the inherent execution time, or *latency*, of instructions; the latency is similar in length to the multicycle approach. Unlike that approach, which uses the same hardware repeatedly during instruction execution, pipelining starts an instruction every clock cycle by having dedicated hardware. Similarly, multiple issue adds additional datapath hardware to allow multiple instructions to begin every clock cycle, but at an increase in effective latency. Figure 6.53 shows the datapaths from Figure 6.52 placed according to the amount of sharing of hardware and **instruction latency**.

Pipelining and multiple issue both attempt to exploit instruction-level parallelism. The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited. Scheduling and speculation, both in hardware and software, are the primary techniques used to reduce the performance impact of dependences.

The switch to longer pipelines, multiple instruction issue, and dynamic scheduling in the mid-1990s has helped sustain the 60% per year processor performance increase that we have benefited from since the early 1980s. In the past, it appeared that the choice was between the highest clock rate processors and the most sophisticated superscalar processors. As we have seen, the Pentium 4 combines both and achieves remarkable performance.

instruction latency The inherent execution time for an instruction.

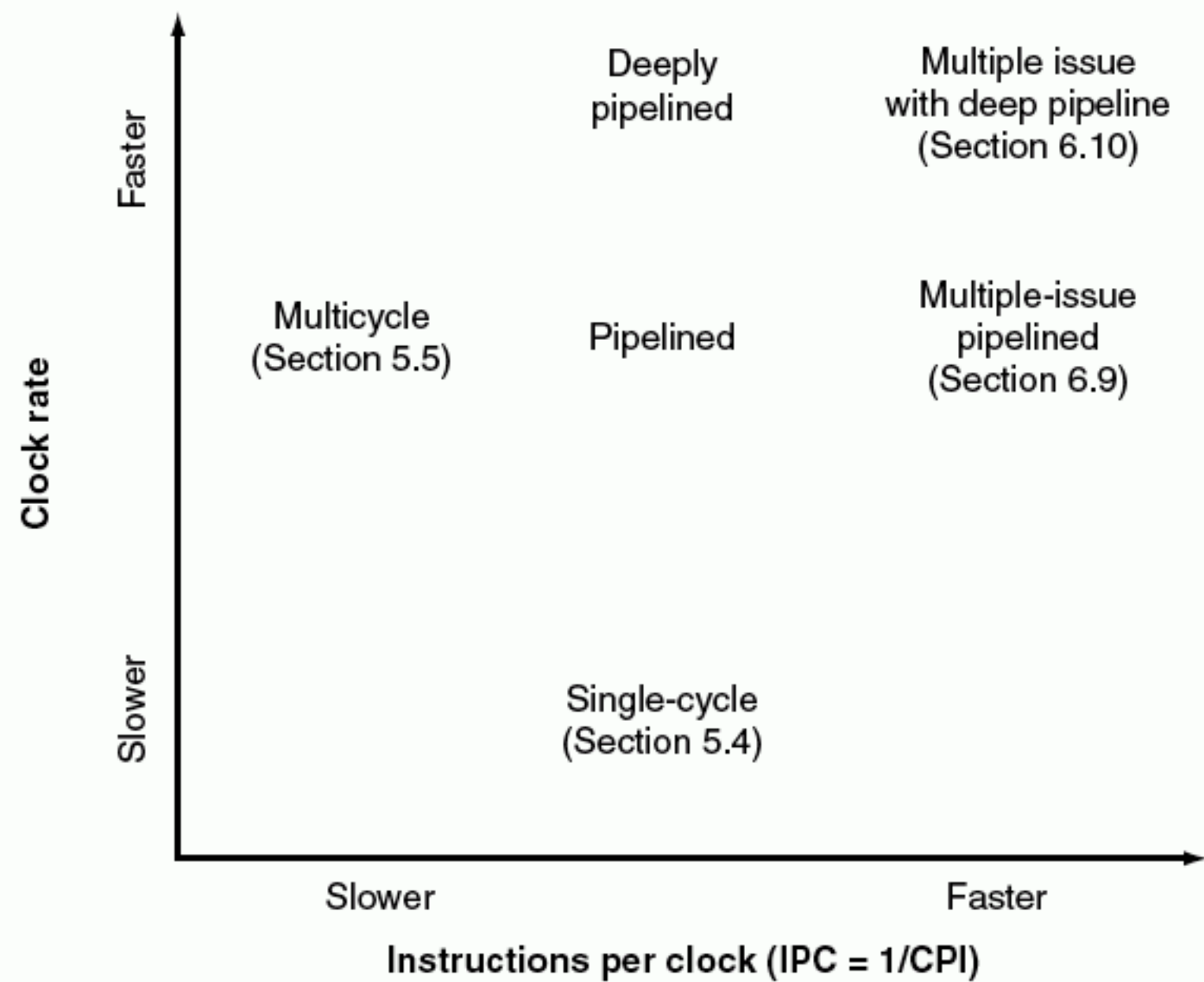


FIGURE 6.52 The performance consequences of simple (single-cycle) datapath and multicycle datapath from Chapter 5 and the pipelined execution model in Chapter 6. Remember that CPU performance is a function of IPC times clock rate, and hence moving to the upper right increases performance. Although the instructions per clock cycle is slightly larger in the simple datapath, the pipelined datapath is close, and it uses a clock rate as fast as the multicycle datapath.

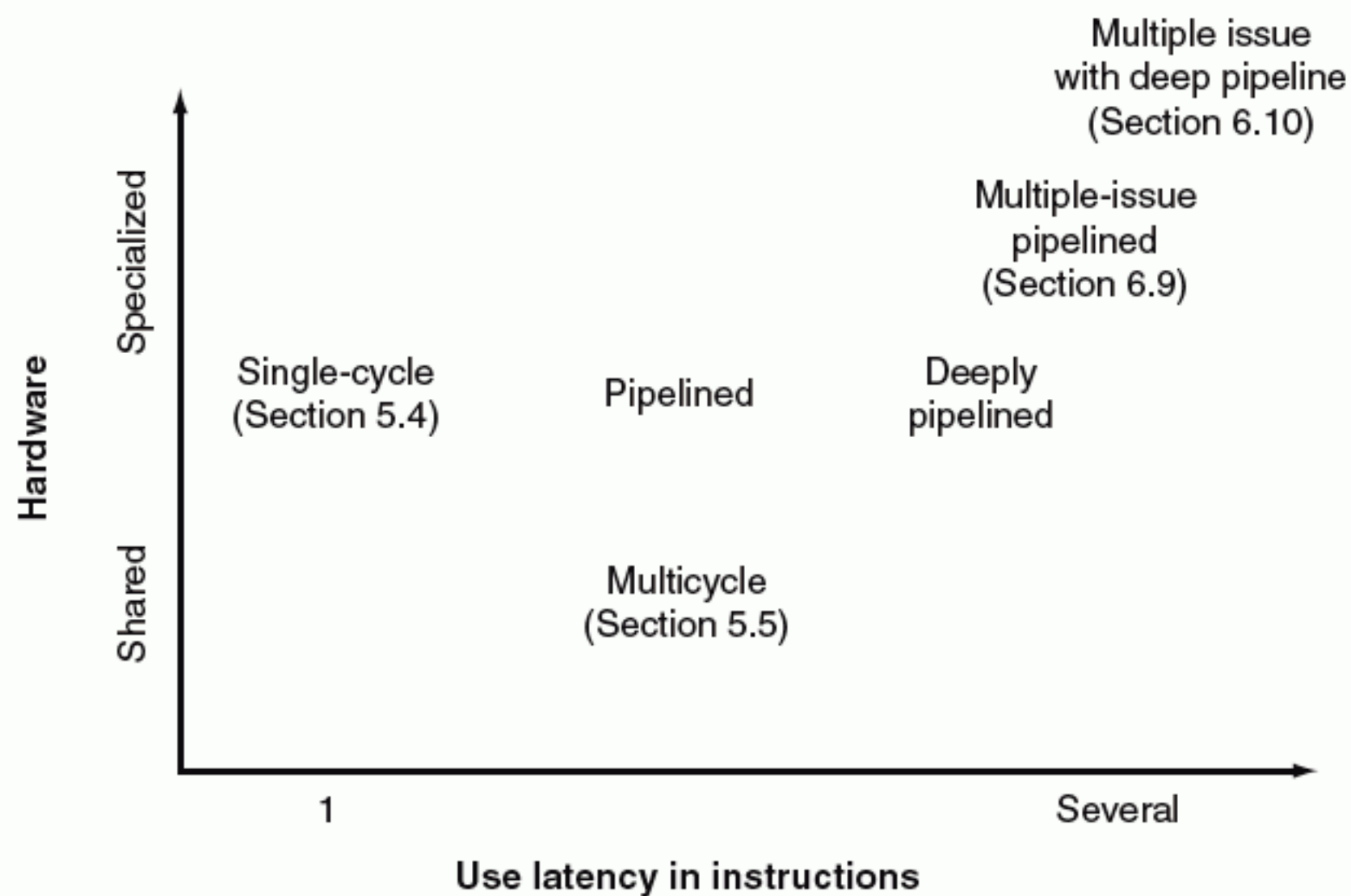



FIGURE 6.53 The basic relationship between the datapaths in Figure 6.52. Notice that the x -axis is use latency in instructions, which is what determines the ease of keeping the pipeline full. The pipelined datapath is shown as multiple clock cycles for instruction latency because the execution time of an instruction is not shorter; it's the instruction throughput that is improved.

With remarkable advances in processing, Amdahl's law suggests that another part of the system will become the bottleneck. That bottleneck is the topic of the next chapter: the memory system.

An alternative to pushing uniprocessors to automatically exploit parallelism at the instruction level is trying multiprocessors, which exploit parallelism at much coarser levels. Parallel processing is the topic of  Chapter 9, which appears on the CD.



Historical Perspective and Further Reading

This section, which appears on the CD, discusses the history of the first pipelined processors, the earliest superscalars, the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.



Exercises

6.1 [5] <§6.1> If the time for an ALU operation can be shortened by 25% (compared to the description in Figure 6.2 on page 373);

- Will it affect the speedup obtained from pipelining? If yes, by how much? Otherwise, why?
- What if the ALU operation now takes 25% more time?

6.2 [10] <§6.1> A computer architect needs to design the pipeline of a new microprocessor. She has an example workload program core with 10^6 instructions. Each instruction takes 100 ps to finish.

- How long does it take to execute this program core on a nonpipelined processor?
- The current state-of-the-art microprocessor has about 20 pipeline stages. Assume it is perfectly pipelined. How much speedup will it achieve compared to the nonpipelined processor?
- Real pipelining isn't perfect, since implementing pipelining introduces some overhead per pipeline stage. Will this overhead affect instruction latency, instruction throughput, or both?

6.3 [5] <§6.1> Using a drawing similar to Figure 6.5 on page 377, show the forwarding paths needed to execute the following four instructions:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

6.4 [10] <§6.1> Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding? Which dependencies are data hazards that will cause a stall?

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

6.5 [5] <§6.1>  [For More Practice](#): Delayed Branches

6.6 [10] <§6.2> Using Figure 6.22 on page 400 as a guide, use colored pens or markers to show which portions of the datapath are active and which are inactive in each of the five stages of the `sw` instruction. We suggest that you use five photocopies of Figure 6.22 to answer this exercise. (We hereby grant you permission to violate the Copyright Protection Act in doing the exercises in Chapters 5 and 6!) Be sure to include a legend to explain your color scheme.

6.7 [5] <§6.2>  [For More Practice](#): Understanding Pipelines by Drawing Them


6.8 [5] <§6.2>  [For More Practice](#): Understanding Pipelines by Drawing Them

6.9 [15] <§6.2>  [For More Practice](#): Understanding Pipelines by Drawing Them

6.10 [5] <§6.2>  [For More Practice](#): Pipeline Registers

6.11 [15] <§§4.8, 6.2>  [For More Practice](#): Pipelining Floating Point

6.12 [15] <§6.3> Figure 6.37 on page 417 and Figure 6.35 on page 415 are two styles of drawing pipelines. To make sure you understand the relationship between these two styles, draw the information in Figures 6.31 through 6.35 on pages 410 through 415 using the style of Figure 6.37 on page 417. Highlight the active portions of the data paths in the figure.

6.13 [20] <§6.3> Figure 6.14.10 is similar to Figure 6.14.7 on page 6.14-9 in the  [For More Practice](#) section, but the instructions are unidentified. Determine as much as you can about the five instructions in the five pipeline stages. If you cannot fill in a field of an instruction, state why. For some fields it will be easier to decode the machine instructions into assembly language, using Figure 3.18 on

page 205 and Figure A.10.2 on page A-50 as references. For other fields it will be easier to look at the values of the control signals, using Figures 6.26 through 6.28 on pages 403 and 405 as references. You may need to carefully examine Figures 6.14.5 through 6.14.9 to understand how collections of control values are presented (i.e., the leftmost bit in one cycle will become the uppermost bit in another cycle). For example, the EX control value for the subtract instruction, 1100, computed during the ID stage of cycle 3 in Figure 6.14.6, becomes three separate values specifying RegDst (1), ALUOp (10), and ALUSrc (0) in cycle 4.

6.14 [40] <§6.3> The following piece of code is executed using the pipeline shown in Figure 6.30 on page 409:

```
lw    $5, 40($2)
add   $6, $3, $2
or    $7, $2, $1
and   $8, $4, $3
sub   $9, $2, $1
```

At cycle 5, right before the instructions are executed, the processor state is as follows:

- The PC has the value 100_{ten} , the address of the `sub_instruction`.
- Every register has the initial value 10_{ten} plus the register number (e.g., register \$8 has the initial value 18_{ten}).
- Every memory word accessed as data has the initial value 1000_{ten} plus the byte address of the word (e.g., `Memory[8]` has the initial value 1008_{ten}).

Determine the value of every field in the four pipeline registers in cycle 5.

6.15 [20] <§6.3>  **For More Practice:** Labeling Pipeline Diagrams with Control

6.16 [20] <§6.4>  **For More Practice:** Illustrating Diagrams with Forwarding

6.17 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
add    $2, $3, $1
sub    $4, $3, $5
add    $5, $3, $7
add    $7, $6, $1
add    $8, $2, $6
```

At the end of the fifth cycle of execution, which registers are being read and which register will be written?

6.18 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the forwarding unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

6.19 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the hazard detection unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

6.20 [20] <§§6.4, 6.5>  **For More Practice:** Forwarding in Memory

6.21 [5] <§6.5> We have a program of 10^3 instructions in the format of "lw, add, lw, add, ..." The add instruction depends (and only depends) on the lw instruction right before it. The lw instruction also depends (and only depends) on the add instruction right before it. If the program is executed on the pipelined datapath of Figure 6.36 on page 416:

- What would be the actual CPI?
- Without forwarding, what would be the actual CPI?


6.22 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
lw    $4, 100($2)
sub   $6, $4, $3
add   $2, $3, $5
```

How many cycles will it take to execute this code? Draw a diagram like that of Figure 6.34 on page 414 that illustrates the dependencies that need to be resolved, and provide another diagram like that of Figure 6.35 on page 415 that illustrates how the code will actually be executed (incorporating any stalls or forwarding) so as to resolve the identified problems.

6.23 [15] <§6.5> List all the inputs and outputs of the forwarding unit in Figure 6.36 on page 416. Give the names, the number of bits, and brief usage for each input and output.

6.24 [20] <§6.5>  **For More Practice:** Illustrating Diagrams with Forwarding and Stalls

6.25 [20] <§6.5>  **For More Practice:** Impact on Forwarding of Moving It to ID Stage

6.26 [15] <§§6.2–6.5>  **For More Practice:** Impact of Memory Addressing Mode on Pipeline

6.27 [10] <§§6.2–6.5>  **For More Practice:** Impact of Arithmetic Operations with Memory Operands on Pipeline

6.28 [30] <§6.5, Appendix C>  **For More Practice:** Forwarding Unit Hardware Design

6.29 [1 week] <§§6.4, 6.5> Using the simulator provided with this book, collect statistics on data hazards for a C program (supplied by either the instructor or with the software). You will write a subroutine that is passed the instruction to be executed, and this routine must model the five-stage pipeline in this chapter. Have your program collect the following statistics:

- Number of instructions executed.
- Number of data hazards not resolved by forwarding and number resolved by forwarding.
- If the MIPS C compiler that you are using issues `nop` instructions to avoid hazards, count the number of `nop` instructions as well.

Assuming that the memory accesses always take 1 clock cycle, calculate the average number of clock cycles per instruction. Classify `nop` instructions as stalls inserted by software, then subtract them from the number of instructions executed in the CPI calculation.

6.30 [7] <§§6.4, 6.5> In the example on page 425, we saw that the performance advantage of the multicycle design was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Find the relative performance of the single-cycle and multicycle designs. In the next few exercises, we extend this to the pipelined design, which requires lots more work!

6.31 [10] <§6.6>  **For More Practice:** Coding with Conditional Moves

6.32 [10] <§6.6>  **For More Practice:** Performance Advantage of Conditional Move

6.33 [20] <§§6.2–6.6> In the example on page 425, we saw that the performance advantage of both the multicycle and the pipelined designs was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Draw the modified pipeline. List all the possible new forwarding situations and all possible new hazards and their length.

6.34 [20] <§§6.2–6.6> Redo the example on page 425 using the restructured pipeline of Exercise 6.33 to compare the single-cycle and multicycle. For branches, assume the same prediction accuracy, but increase the penalty as appropriate. For loads, assume that the subsequent instructions depend on the load with a probability of 1/2, 1/4, 1/8, 1/16, and so on. That is, the instruction following a load by two has a 25% probability of using the load result as one of its sources. Ignoring any other data hazards, find the relative performance of the pipelined design to the single-cycle design with the restructured pipeline.

6.35 [10] <§§6.4–6.6> As pointed out on page 418, moving the branch comparison up to the ID stage introduces an opportunity for both forwarding and hazards that cannot be resolved by forwarding. Give a set of code sequences that show the possible

forwarding paths required and hazard cases that must be detected, considering only one of the two operands. The number of cases should equal the maximum length of the hazard if no forwarding existed.

6.36 [15] <§6.6> We have a program core consisting of five conditional branches. The program core will be executed thousands of times. Below are the outcomes of each branch for one execution of the program core (T for taken, N for not taken).

Branch 1: T-T-T
 Branch 2: N-N-N-N
 Branch 3: T-N-T-N-T-N
 Branch 4: T-T-T-N-T
 Branch 5: T-T-N-T-T-N-T

Assume the behavior of each branch remains the same for each program core execution. For dynamic schemes, assume each branch has its own prediction buffer and each buffer initialized to the same state before each execution. List the predictions for the following branch prediction schemes:

- Always taken
- Always not taken
- 1-bit predictor, initialized to predict taken
- 2-bit predictor, initialized to weakly predict taken

What are the prediction accuracies?

6.37 [10] <§§6.4–6.6> Sketch all the forwarding paths for the branch inputs and show when they must be enabled (as we did on page 407).

6.38 [10] <§§6.4–6.6> Write the logic to detect any hazards on the branch sources, as we did on page 410.

6.39 [10] <§§6.4–6.6> The example on page 378 shows how to *maximize* performance on our pipelined datapath with forwarding and stalls on a use following a load. Rewrite the following code to *minimize* performance on this datapath—that is, reorder the instructions so that this sequence takes the *most* clock cycles to execute while still obtaining the same result.

```
lw    $2, 100($6)
lw    $3, 200($7)
add   $4, $2, $3
add   $6, $3, $5
sub   $8, $4, $6
lw    $7, 300($8)
beq   $7, $8, Loop
```


6.40 [20] <§6.6> Consider the pipelined datapath in Figure 6.54 on page 461. Can an attempt to flush and an attempt to stall occur simultaneously? If so, do they result in conflicting actions and/or cooperating actions? If there are any cooperating actions, how do they work together? If there are any conflicting actions, which should take priority? Is there a simple change you can make to the datapath to ensure the necessary priority? You may want to consider the following code sequence to help you answer this question:

```

        beq $1, $2, TARGET    # assume that the branch is taken
        lw  $3, 40($4)
        add $2, $3, $4
        sw  $2, 40($4)
TARGET:  or  $1, $1, $2

```

6.41 [15] <§§ 6.4, 6.7> The Verilog for implementing forwarding in Figure 6.7.2 on page 6.7-4–6.7-5 did not consider forwarding of a result as the value to be stored by a SW instruction. Add this to the Verilog code.

6.42 [5] <§§6.5, 6.7> The Verilog for implementing stalls in Figure 6.7.3 on page 6.7-6–6.7-7 did not consider forwarding of a result to use in an address calculation. Make this simple addition to the Verilog code.

6.43 [15] <§§6.6, 6.7> The Verilog code for implementing branch hazard detection and stalls in Figure 6.7.3 on page 6.7-6–6.7-7 does not detect the possibility of data hazards for the two source registers of a BEQ instruction. Extend the Verilog in Figure 6.7.3 on page 6.7-6–6.7-7 to handle all data hazards for branch operands. Write both the forwarding and stall logic needed for completing branches during ID.

6.44 [10] <§§6.6, 6.7> Rewrite the Verilog code in 6.7.3 on page 6.7-6–6.7-7 to implement a delayed branch strategy.

6.45 [20] <§§6.6, 6.7> Rewrite the verilog code in Figure 6.7.3 on page 6.7-6–6.7-7 to implement a branch target buffer. Assume the buffer is implemented with a module with the following definition:

```

module PredictPC (currentPC,nextPC,miss,update,destination);
    input currentPC,
        update, // true if previous prediction was unavailable or incorrect
        destination; // used with update to correct a prediction
    output nextPC, // returns the next PC if prediction is accurate
        miss; // true means no prediction in buffer
endmodule;

```

Make sure you accomodate all three possibilities: a correct prediction, a miss in the buffer (that is, miss = true), and an incorrect prediction. In the last two cases, you must also update the prediction.